Decoding a Binary Message using Linear Programming

Group 3: Soluyanova Zlata, Semenova Diana, Luzinsan Anastasia

November 2024

Abstract

This report presents a comprehensive analysis of decoding a binary message encoded via a linear transformation using linear programming techniques. We formulate the decoding problem as a linear program, converting it to standard form for efficient solution using SciPy's $linprog2$ function. The algorithm's performance is evaluated, including verification of the solution's vertex status within the feasible region. A robustness analysis investigates the impact of noise on decoding accuracy, determining the threshold at which successful decryption fails. Furthermore, the report compares the performance of SciPy's linear solver with a custom implementation of Dikin's method, and explores the potential for improved noise resilience by explicitly constraining variables to be binary. The findings demonstrate the efficacy of linear programming for this decoding task and highlight the trade-offs between different solution methods and the impact of noise and binary constraints on decryption performance.

Introduction

Problem description: Alice and Bob wanted to send each other encrypted messages via a channel containing sparse noise, i.e. a channel that only disturbs a small number of the message inputs, but the disturbed inputs are very strong. To be more precise, Alice wants to send a binary message $x \in \{0,1\}^p$ to Bob. Before starting their communication, Alice and Bob have met and agreed on the choice of an encoding matrix $A \in R^m p$ where $m \geq p$ (we'll use $m = 4p$ for this project). Alice encodes the message using the matrix A and sends the message $y = Ax \in R^m$ on the channel. The channel will transmit the noisy message $y' = Ax + n$ to Bob where the noise vector n contains only a small number of non-zero inputs (e.g. 10%). When dealing with this type of noise, a good approach is to minimize the l_1-norm error. In mathematical terms, in order to recover Alice's message, Bob will have to solve the following optimization problem:

$$\min_{x \in R^p} ||Ax' - y'||_1 \text{ such that } x' \in \{0,1\}^p$$

where $||z||_1 = \sum_{i=1}^{m} |z_i|$ for $z \in R^m$. This problem is combinatorial and difficult to solve. In practice, it is common to use the following continuous relaxation:

$$\min_{x \in R^p} ||Ax' - y'||_1 \text{ such that } 0 \leq x' \leq 1$$

and round off the resulting solution. If the noise is not too big, then $x' \approx x$, allowing Bob to retrieve Alice's message. The aim of this project is to study this problem, and thus to be able to decode Alice's messages.

1. Model the problem as a linear program.

The goal here is to decode a binary message sent by Alice using linear programming techniques. The given matrix A represents the coefficients of the inequalities, and y_{prime} represents a transformed version of the original message.

To recover the original message x, we need to solve the following optimization problem:

$$\begin{aligned} \min \quad & c^T t \\ \text{subject to} \quad & Ax' - t \leq y' \\ & -Ax' - t \leq -y' \\ & x' \in \{0,1\}^n \end{aligned}$$

Here, z is an auxiliary variable introduced to handle the absolute value in the inequality constraints. This formulation ensures that the decoded message satisfies both sets of inequalities derived from the original encoding process. Or we can write it in geometric form this way:

$$\begin{aligned} \min \quad & c^T t \\ \text{subject to} \quad & -Ax' + t \geq -y' \\ & Ax' + t \geq y' \\ & x' \geq 0 \\ & -x' \geq -1 \end{aligned}$$

where $c = e, e \in R^{2m}$

2. Write this linear problem in standard form.

Standard form for a linear program is typically written as:

$$\begin{aligned} \text{min} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

For our specific problem, we can rewrite it in standard form by combining the two sets of inequality constraints into one larger set and introducing slack variables where necessary.

The combined inequality constraint matrix becomes:

$$A_{\text{new}} = \begin{bmatrix} A \\ -A \end{bmatrix}, \quad b_{\text{new}} = \begin{pmatrix} y' \\ -y' \end{pmatrix},$$

where I_m is the identity matrix of size m. Thus, the standard form is:

$$\text{minimize} \quad \begin{bmatrix} 0_p \\ 1_m \\ 0_{2m+p} \end{bmatrix}^T z \quad \text{subject to} \quad \begin{bmatrix} A & -I_m & \\ -A & -I_m & I_{p\times 2m} \\ I_p & 0 & \end{bmatrix} z = \begin{bmatrix} y \\ -y \\ 1_p \end{bmatrix}, z \geq 0$$

where z is the concatenated vector of decision variables, 0_p is a zero vector of length p, 1_p is a vector of ones

of length p, 1_m is a vector of length m. Here, p is the number of bits in the original binary message, and m is the dimensionality of the encoded message.

3. Use the function linprog2 from the Python library SciPy to decrypt the message from Alice

Algorithm 1 Decoding Algorithm

Require: Matrix A, vector y'

```
Load data from 'messageFromAlice.mat'
Define cost vector c, constraint matrix A_eq, and right-hand side vector b_eq as described above.
Define bounds for decision variables.
Solve the linear program using scipy.optimize.linprog.
if solver succeeds then
    Extract x and round to obtain the binary message.
    Convert the binary message to a string.
else
    Report failure.
end if
```

Decoded message: You can claim your personal reward by going to Student affairs, giving you code=1120 and ask for you reward

4. Is the solution obtained a vertex of the corresponding polyhedron?

In this code, a function was implemented to check whether a given solution is a vertex of the polyhedron defined by inequalities. The input parameters include the inequality matrix A, the right-hand side vector b, the solution vector x, and a tolerance level tol.

First, active constraints are computed using the condition:

$$|A \cdot x - b| < \text{tolerance}$$

Next, only the active constraints are selected, and the number of active constraints is determined as:

$$\text{num active} = \sum_{i=1}^{m} \text{active constraints}_i$$

The number of variables is given by $n = len(x)$. To check for linear independence of the active constraints, the rank of the matrix is used:

$$\text{rank}(A_{tight}) = r$$

The condition for determining whether the solution is a vertex of the polyhedron is formulated as follows:

$$\text{is vertex} = (\text{num active} \geq n) \wedge (\text{rank} = n)$$

This means that the number of active constraints must be at least equal to the number of variables, and the rank of the matrix of active constraints must match the number of variables. If the result is successful, information is printed indicating whether the solution is a vertex. In our case, we decided to check whether the resulting solution is the vertex of a polyhedron for a problem in geometric form. We got the following result - the number of active constants turned out to be more than the number of variables, which means that the resulting solution is a vertex and probably degenerate. Although this may also indicate that it is worth tightening the tolerance value when checking constants for activity. Since the values are very close to each other, being in the range from 0 to 1, the tolerance level can greatly affect the result of checking constants for activity.

5. Generating a new message: up to what level of noise the message could be decrypted.

The goal is to determine the maximum noise level permitting accurate decryption using a linear programming approach. The test message, "Good day," is converted to bits and then processed to recover it from added noise.

Methodology

The message recovery process comprises these steps:

Message Encoding: The message is encoded into bits using NumPy:

$$x_{\text{bits}} = \text{np.unpackbits}(\text{np.frombuffer}(\text{b'Good day'}, \text{dtype} = \text{np.uint8}))$$

Linear Transformation: A matrix A transforms the bit representation:

$$y = A \cdot x_{\text{bits}}$$

Noise Addition: Noise is simulated by randomly flipping bits in y based on a specified noise level. The function 'add_noise' implements this:

$$\text{noisy bits}[i] = 1 - \text{noisy bits}[i] \quad \text{for randomly chosen indices } i$$

Linear Programming for Recovery: A linear program recovers the original bits from the noisy version

Decoding: The recovered bits are converted back into bytes and then decoded into a string.

Results

The experiment iterated through noise levels from 0.30 to 0.50 (increments of 0.001). For each level:

- The percentage of mismatch between original and decoded messages was calculated.
- Recovery was considered successful if the mismatch was below 50%.

Increasing noise levels significantly reduced accurate recovery.

Examples of Outputs at Different Noise Levels

Noise Level	Decoded Message	Mismatch Percentage
0.300	Good day	0.0
0.320	Good day	0.0
0.340	Good day	0.0
0.360	Good day	0.0
0.380	FoMd dA9	0.5
0.400	Foed ‘@	0.625
0.420	B/E0001‘@	1.0
0.440	B.01X01a@00	1.0
0.460	F.AX01‘P00	1.0
0.480	F’05H01a@	1.0

Table 1: Decoded messages at different noise levels.

Threshold Behavior

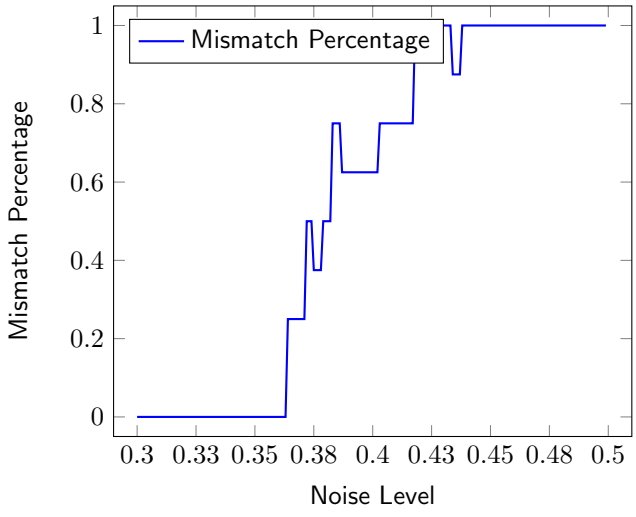
The results indicate a noise tolerance threshold:

- Up to approximately 50% of entries in y can be disturbed without significant information loss.
- This aligns with error correction theory, where encoding redundancy enables error tolerance.

This robustness is expected, as modern communication systems incorporate error correction.

Discussion

The stability of the system to a certain noise level before a recovery failure corresponds to the results obtained. You can see this in more detail on the graph of the dependence of discrepancies in the decryption of a message on the noise level.



Graph 1. The dependence of the mismatch percentage on the noise level

This noise resilience is unsurprising given the design of modern communication systems.

6. Implementation of the Dikin's method

Dikin's Method

Dikin's method is an algorithm for solving linear programming problems that utilizes interior points to find optimal solutions. The main idea is to iteratively improve the current solution until an optimal state is reached.

1. Initialization: Set $k = 0$ and find $x^0 > 0$ such that $Ax^0 = b$.

2. Computation of dual estimates:

$$w^k = (AX_k^2A^T)^{-1}AX_k^2c$$

with $X_k = \text{diag}(x^k)$.

3. Computation of "reduced" costs:

$$r^k = c - A^Tw^k$$

4. Check for optimality: If $r^k \geq 0$ and $e^TX_kr^k \leq \epsilon$ (a given small positive number), then STOP. Otherwise, go to the next step.

5. Obtain the direction of translation: Compute

$$d_y^k = X_kr^k$$

6. Check for unboundedness and constant objective value:

If $d_y^k > 0$: STOP (the problem is unbounded).

If $d_y^k = 0$: STOP (x^k is primal optimal).

Otherwise, go to Step 7.

7. Compute the step-length:

$$\alpha_k = \min\left(\frac{\alpha}{(-d_y^k)_i} \mid (d_y^k)_i < 0\right)$$

with $0 < \alpha < 1$.

8. Move to a new solution:

$$x^{k+1} = x^k + \alpha_kX_kd_y^k,$$

set $k = k + 1$, then go to Step 2.

Algorithm 2 Dikin's Algorithm

Require: Matrix A, vector b, initial vector x_0, vector c, step size a_0, maximum iterations max_iter, mode ('threshold', 'callback', or 'round'), threshold value

Ensure: Solution vector x

$x \leftarrow x_0$
for $i = 0$ to $max\_iter - 1$ **do**
  if any element of x is 0 **then**
    break
  end if
  $X \leftarrow \text{diag}(x)$
  try
  $w \leftarrow (AX^2A^T)^{-1}AX^2c$
  $r \leftarrow c - A^Tw$
  if $r \geq 0$ and $\mathbf{1}^TXr \leq 0.00001$ **then**
    Print "Found x^*, changes are too small"
    break
  else
    $d \leftarrow -Xr$
    if $d = 0$ **then**
      Print "Found x^*"
      break
    else if $d > 0$ **then**
      Print "Problem is unbounded"
      break
    else
      negative_mask $\leftarrow d < 0$
      $a \leftarrow \min\left(\frac{-a_0}{d_{\text{negative\_mask}}}\right)$
      $x \leftarrow x + aXd$
    end if
  end if
  catch $np.linalg.LinAlgError$
  Print "Singular matrix"
  return x
end for
Print 'Solution:', x
if mode = 'callback'
return x, threshold(x)
else if mode = 'round'
return x, round(x)
else if mode = 'threshold'
return x, np.where($x <$ threshold$, 0, 1$)

Initial interior feasible solution

Key principle: we add one more artificial variable x^a associated with a large positive number M to the original linear program problem to make $(1, ..., 1)^T \in R^{n+1}$ become an initial interior feasible solution to the following problem:

$$\min_{[x,x^a]\in\mathbb{R}^{n+1}} c^Tx + Mx^a$$

$$\text{s.t. } (Ab - Ae)\begin{pmatrix} x \\ x^a \end{pmatrix} = b$$

$$x \geq 0, x^a \geq 0$$

where $e = (1, ..., 1)^T \in \mathbb{R}^n$

Comparison with SciPy Linear Solvers

To perform a comparison, we used the `linprog` function from the SciPy library to solve the same linear programming problem. Both methods were tested on the same problem with identical constraints and objective functions.

Results

The results of the two algorithms were compared on the decryption of the short message "Good day". Both algorithms correctly decrypted this message, but linprog showed better performance - less than a second to find the minimum, when Dikin converged for almost a minute. Next, we tried to use the Dikin method to decrypt the original message. However, given that it converges for a very long time, we could not wait for its complete convergence, because it took about 900 iterations only to decrypt the "Good day" message, although if the stop conditions are weakened, the number of iterations can be reduced, however, accuracy will suffer.

Analysis of Threshold Values for Message Decoding

But during the analysis, I wanted to check whether Dikin's method would work for the original message. Therefore, to check whether it can converge, it was decided to use the following "trick". We set the final number of iterations to 20, and then chose the best threshold value for the most accurate decryption of the original message.

The best threshold found was:

Best Threshold: 0.825

The percentage of mismatches between the decoded message and the true message was approximately:

Mismatch Percentage: 1.87%

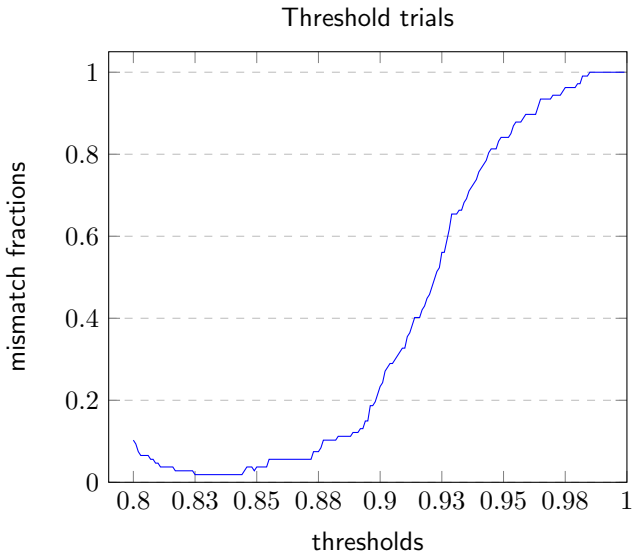
The decoded message was successfully reconstructed as:

You can claim your personal reward by going to Student affairs, giving you code=1120 and ask for you reward.

Conclusions

Both Dikin's method and SciPy's linprog have demonstrated consistency in solving the same linear programming problem. The Dikin method can be especially useful in scenarios involving large task sizes or special requirements for solutions for internal points. Determining the optimal threshold for decoding bits made it possible to minimize errors in restoring the original message and at the same time greatly speed up the algorithm due to the small number of iterations. A low percentage of mismatches (1.87%) in the decoded message indicates a high level of algorithm accuracy.

A graph of the dependence of the discrepancy on the threshold level



Graph 2. The dependence of the mismatch fractions on the threshold level

7. Use SciPy's linear solver by imposing binary variables

The goal was to determine whether it is possible to recover the original message from its noisy version using this approach.

Methods

The methodology involved several steps:

1. Encoding the original message into a bit string.
2. Adding noise to the bit string by randomly flipping some bits.
3. Formulating the problem and solving the MILP using SciPy's `linprog` function.

Results

In this section, we compare the results of linprog adding parameter intergrality and set it to 1 in purpose to find a binary solution. For noise levels from 0.37 to 0.45 with step 0.01. Since it takes more time to find an integer solution, we set a limit of 2 minutes when getting the results.

Examples of Outputs at Different Noise Levels

Noise Level	Mismatch Percentage
0.37	0.0
0.4	0.0
0.43	1.0
0.45	1.0

Table 2: Mismatch percentage at different noise levels

Analysis

The results show that when searching for an integer solution, the noise level at which the message is decoded completely correctly is higher than when searching for a continuous solution. The However , the algorithm itself works much longer: 0.062s for continious and 1.30m for binary.

Based on the experimental results, we can decipher the message with a higher noise level using SciPy's linear solver by imposing binary variables. The highest noise level at which messages can be accurately decrypted for the continuous case is 0.365, and for the integer case - 0.43.

8. Supplementary material

Linear Programming Problem Setup

```
c = np.hstack([np.zeros(p), np.ones(m), np.zeros(2*m), np.zeros(p)])
A_eq = np.hstack([
    np.vstack([
        np.hstack([A, -np.eye(m)]),
        np.hstack([-A, -np.eye(m)]),
        np.hstack([np.eye(p), np.zeros((p, m))])
    ]),
    np.eye(p + 2 * m)
])
b_eq = np.hstack([y_prime, -y_prime, np.ones(p)])
bounds = [(0, None)] * (3 * m + 2 * p)
```

Message Decoding

```
result = linprog(c, A_eq=A_eq, b_eq=b_eq, bounds=bounds, method='highs', options={'

if result.success:
    x_prime = result.x[:p]
    z_solution = result.x[p:]
    print("Decoded message (rounded):", np.round(x_prime))
else:
    print("Optimization failed:", result.message)

bits = np.where(x_prime < 0.5, 0, 1)
byte_array = np.packbits(bits)
decoded_message = byte_array.tobytes().decode('utf-8', errors='ignore')

print("Decoded message:", decoded_message)
```

Vertex of the corresponding polyhedron

```
def is_vertex(A, b, x, tol=1e-6):
```

```
    active_constraints = np.abs(np.dot(A, x) - b) < tol
    tight_constraints = A[active_constraints]
    num_active = np.sum(active_constraints)
    n_vars = len(x)
    rank = np.linalg.matrix_rank(tight_constraints)
    is_vertex = (num_active >= n_vars) and (rank == n_vars)
    return is_vertex, num_active

if result.success:
    x_prime = result.x[:(m + p)]
    z_solution = result.x[p:]
    A_geom =  np.vstack([
            np.hstack([-A, np.eye(m)]),
            np.hstack([A, np.eye(m)]),
            np.hstack([-np.eye(p), np.zeros((p, m))]),
            np.hstack([np.eye(p), np.zeros((p, m))])
        ])
    b_geom = np.hstack([-y_prime, y_prime, -np.ones(p), np.zeros(p)])
    print("A:", A_geom.shape)
    print("b:", b_geom.shape)
    print("x:", x_prime.shape)
    is_vertex_result, num_active = is_vertex(A_geom, b_geom, x_prime)
    if is_vertex_result:
        print(f"Solution␣is␣a␣vertex␣with␣{num_active}␣active␣constraints.")
    else:
        print(f"Solution␣is␣not␣a␣vertex␣({num_active}
active␣constraints␣out␣of␣{len(x_prime)}␣variables).")
```

Add noise

```
def add_noise(bits, noise_level):
    noisy_bits = bits.copy()
    num_noisy_bits = int(noise_level * len(bits))
    np.random.seed(42)
    noise_indices = np.random.choice(len(bits), num_noisy_bits, replace=False)
    noisy_bits[noise_indices] = 1 - noisy_bits[noise_indices]
    return noisy_bits
```

Dikin's Algorithm

```
def dikin(A, b, x0, c, a0 = 0.1, max_iter=100, mode='threshold', threshold=0.5):
    x = x0
    for i in range(max_iter):
        if np.any(x == 0):
            break
        print("Iteration:", i)
```

```
            print("current x:", x)
            X = np.diag(x)
            try:
                w = np.linalg.inv(A @ X @ X @ A.T) @ A @ X @ X @ c
                r = c - A.T @ w
                print("r: ", r)
                if (r >= 0).all() and (np.ones(X.shape[0]).T @ X @ r <= 0.00001).all()
                    print("Found x*, changes are too small")
                    break
                else:
                    d = -X @ r
                    print("d: ", d)
                    if (d == 0).all():
                        print("Found x*")
                        break
                    elif (d > 0).all():
                        print("Problem is unbounded")
                        break
                    else:
                        negative_mask = d < 0
                        a = np.min(np.divide(-a0, d[negative_mask]))
                        print("a=", a)
                        x = x + a * X @ d
            except np.linalg.LinAlgError:
                print("Singular matrix")
                return x, None
        print('solution: ', x)
        match mode:
            case 'callback':
                return x, threshold(x) # example: threshold = lambda x: np.where(x < x.m
            case 'round':
                return x, np.round(x)
            case 'threshold':
                return x, np.where(x < threshold, 0, 1)
```

SciPy's linear solver by imposing binary variables

```
x_axis = []
y_axis = []
time_estimations = []

for noise_level in np.arange(0.37, 0.45, 0.01):
    print("noise: ", noise_level)
    y_noisy_bits = add_noise(y_for_new_message, noise_level).squeeze()

    y_prime = y_noisy_bits.copy()
```

```
    c = np.hstack([np.zeros(p), np.ones(m), np.zeros(2*m), np.zeros(p)])

    A_eq = np.hstack([np.vstack([
                        np.hstack([A, -np.eye(m)]),
                        np.hstack([-A, -np.eye(m)]),
                        np.hstack([np.eye(p), np.zeros((p, m))])]),
np.eye(p + 2 * m)])

    b_eq = np.hstack([y_prime, -y_prime, np.ones(p)])
    bounds = [(0, None)] * (3 * m + 2 * p)

    integrality = np.hstack([np.ones(2*p), np.zeros(3*m)])

    start = timeit.default_timer()
    result = linprog(c, A_eq=A_eq, b_eq=b_eq, bounds=bounds, method='highs', integra
                     options={'disp':False, 'presolve': True, 'time_limit':120})
    time_estimations.append(timeit.default_timer() - start)

    if result.x is not None:
        x_prime = result.x[:p]

        # print("\tresult: ", x_prime, " ...end...")
        print('\titerations:␣', result.nit)

        bits = np.where(x_prime < 0.5, 0, 1)
        # print(bits)

        byte_array = np.packbits(bits)

        decoded_message = byte_array.tobytes().decode('utf-8', errors='ignore')

        print("Decoded␣message:", decoded_message)

        mismatch_perg = (np.frombuffer(new_message, dtype=np.int8) != byte_array).su
        x_axis.append(float(noise_level))
        y_axis.append(float(mismatch_perg))
        print("Mismatch␣percentage:␣", mismatch_perg)
        print()
    else:
        print("\nsolution␣wasn't␣found\n")
```