Mini-project in Topics in Error Correcting Codes

in Locally decodable codes

Name : Rola Dabbah
ID : 316435247
Department : Computer Science
Submission Date : 26.01.2020
Chosen paper : High Rate LDC - High-rate codes with sublinear-time decoding .

Table of contents:

Section	Pages
1. About the project	3
2.Introduction	3
3.Problem Statement	3-4
4.Code Objective	5
5.Reed-Muller Code	5-8
6.Implementation	9-12
7.Running Examples	13-14

1. About The Project:

The algorithm that I implemented in this project is based on the article:
High-rate codes with sublinear-time decoding .
ACM Reference Format:Swastik Kopparty, Shubhangi Saraf, Sergey Yekhanin, 2014. High-rate codes with sublinear-time decoding.ACM Trans. Embedd. Comput. Syst.9, 4, Article 39 (March 2010).

Link of the paper :
http://nebula.wsimg.com/124497399ace322088695d632cc18539?AccessKeyId=0EF19C92671ED94CE585&disposition=0&alloworigin=1

2. Introduction :

In transmitting data from one parity to another , a noise in the communication can occur , means that the sequences of Σ (the “alphabet” of the code) - called code words -that are transmitted needs to be checked if its in the dictionary (allowable words) of the relevant codewords of the code .
After transmission over a noisy channel, we can check to see if the received sequence is in the dictionary of code words and if not, choose the codeword most similar to what was received, by using **Error Correcting Codes.**

Example :

- 011,100,001 : id code with 3 code words, each with 3 bits.
- according to this code , 010 is a illegal word.

3. Problem Statement :

In other words, we have a set Σ (the “alphabet”), and we want to construct a subset C (the “code”) of Σ^n , of size $|\Sigma|^k$ (we call k the “message length”), with the following local correction property: given access to any $r \in \Sigma^n$ which is close to some code word $c \in C$, and given $i \in [n]$, it is possible to make few queries to the coordinates of r, and with high probability output c_i.
As a solution, we can use classical Error-Correcting Codes that allow one to encode a k-bit message x into an n-bit codeword C(x), in such a way that x can still be recovered even if C(x) gets corrupted in a number of coordinates.

Error correcting codes (ECC) :

ECC is a code that checks read or transmitted data for errors and corrects them as soon as they are found. It‘s similar to parity checking except that it corrects errors immediately upon detection. ECC is becoming more common in the field of data storage and network transmission hardware, especially with the increase of data rates and corresponding errors , It can be used for error detection.
Error detection - A code with *minimum* Hamming distance, *d*, can detect up to *d* − 1 errors in a code word.

- **Hamming distance** between two strings of equal length is the number of positions at which the corresponding symbols are different. In other words, it measures the minimum number of *substitutions* required to change one string into the other, or the minimum number of *errors* that could have transformed one string into the other.

(Using minimum-distance-based error-correcting codes for error detection can be suitable if a strict limit on the minimum number of errors to be detected is desired).

Example : Codes with minimum Hamming distance *d* = 2 are degenerate cases of error-correcting codes, and can be used to detect single error

Error correction - A code with *minimum* Hamming distance, *d*, can correct up to *d* − 1 / 2 errors in a code word. Because given a code with distance d and two words A and B where distance(A, B) = d then if there is d / 2 errors in received word W when transmitting A, then W may be recovered to the original code word A or B. Therefore, W cannot be recovered correctly. The traditional way to recover information about x given access to a corrupted version of C(x) is to run a decoder for C, which would read and process the entire corrupted codeword, and then recover the entire original message x. Suppose that one is only interested in recovering a single bit or a few bits of x. In this case, codes with more efficient decoding schemes are possible, allowing one to read only a small number of coordinates. Such codes are known as Locally Decodable Codes(LDCs).

Locally decodable codes are error-correcting codes that allow reconstruction of an arbitrary bit xi,by looking only at t<<k randomly chosen coordinates of (a possibly corrupted) C(x).

The main parameters of a locally decodable code that measure its utility are the code-word length n (as a function of the message length k) and the query complexity of local decoding. The length measures the amount of redundancy that is introduced into the message by the encoder. The query complexity counts the number of bits that need to be read from a (corrupted) codeword in order to recover a single bit of the message.Ideally, one would like to have both of these

parameters as small as possible. One however cannot minimize the codeword length and the query complexity simultaneously , there is a trade-off.
As a result, we next define our Objective.

4. Code Objective :

Our **goal** is to construct such a subset C with **rate** k/n large (the length (k) of the original word x is not too small comparing with the length (n) of the encoded word C(x)) and to allow local decoding algorithm with arbitrary polynomially-small time and **query complexity.** One of the popular locally decodable codes is Reed-Muller code.

5. Reed Muller Codes:

The definition of the Reed-Muller**(d, m, q)** code (in general) :
Let q be a prime power, Given a message x= (x1,x2,…,xk), Consider a **m**-variate polynomial p(z1,z2,…, zm) of degree **d** with coefficients a1 , a2 , … , ak where k = (m+t choose t).
The Reed-Muller code is given by evaluations of p at n = | F_q^m | different points, each in F_q^m , where Fq is the finite field over q. The code is C(x) is thus an **m**-tuple (p(α1) , p(α2) , … , p(αn)). In this project we are going to implement a sub-case of reem-muller code where the degree is d and m=2. Its denoted, **Bivariate Reed Muller codes - Reed-Muller(d, 2)**

Bivariate Reed-Muller codes : Let q be a prime power, let δ >0 and let d = (1−δ)q. The Reed-Muller code of degree d bivariate polynomials over F_q (the finite field of cardinality q) is the code defined as follows:
The coordinates of the code are indexed by elements of F_q^2 , and so n= q^2 .

Encoding:
The local encoding problem can be specified as follows:

- Input: a ∈ Fq^2
- Output: C ∈ Fq^{q^2} , when c = P(a)

The codeword corresponding the polynomial P(X,Y) is the vector

C(P) = 〈P(a)〉 (a)∈ Fq^2 ∈ Fq^{q^2} .

The codewords are indexed by bivariate polynomials of degree at most d over the field Fq .
Algorithm:

1. get the polynomial with all the monomials with degree <= d , assign poly
2. $\forall x,y\ ,\ x,y \in [0,q-1]$:
   eval = poly.eval(x,y)
   code word =code word + eval
   -> we evaluate the polynomial "poly" on all the values of x,y in the range [0,q-1] and append it to some variable that saves the the evaluation values (code word) .
3. return code word

Complexity:

- generating the monomials $x^i y^j$ such that $0 \le i+j \le d$, $i,j \in [0,q-1]$ to get the polynomial = $O(q^2)$.
- $\forall x,y\ ,\ x,y \in [0,q-1]$ evaluate the polynomial with x and y = $O(q^2 * K)$. (evaluation of the polynomial -> O(K))
- Total Complexity = $O(q^2 * K)$.

Correction:

The local correction problem can now be specified as follows:

- Input: A point a $\in F^m$ and query access to received word P (contains evaluations of polynomial p, possibly with errors).

- Output: The evaluation p(a)∈F.

Algorithm:

1. choose a random $b \in F^2$. Define a function $\ell : \mathbb{N} \to F^2$ as :
   $\ell(t) = a + t * b$
2. Define $q(t) = p(\ell(t))$. Observe that the $deg(q) = deg(p) = d$ **.
3. Evaluate $p(\ell(1)), p(\ell(2)), \ldots , P(\ell(d+1))$ by querying the received word at points $\ell(1), \ell(2), \ldots , \ell(t+1)$.
4. Since, $q(i) = p(\ell(i))$, we have d+1 evaluations of q at points 1,2,...,d+1. Since, deg(q) = d+1, we interpolate to get the polynomial $q(t)$.
5. Putting t=0 in $q(t)$, we get $p(\ell(0)) = p(a)$.

* Number of queries made by the algorithm is d+1.

** $deg(q) = deg(p) = d$, because when we restrict P(x,y) by $a + t * b$

we assign instead of X a linear function of t and another one instead of Y. Therefore, the degree of (deg (X) + deg(Y)) = deg(t) <= d.

Complexity:

- Number of queries made by the algorithm is d+1.
- Number of evaluations is d+1 at points 1,2,..,d+1 - each evaluation gives O(1).
- Since the complexity of the interpolation is $O((d+1)^2)$ => $O((d+1)^2)$
- Total $O((d+1)^2)$ time complexity.

In a more detailed way, Given a received word r $\in Fq^{q^2}$ such that r is close in Hamming distance to the codeword corresponding to P(X,Y), Given a coordinate a$\in Fq^2$, we want to recover the "corrected" symbol at coordinate a, namely P(a).

Decoding:

The local decoding problem can be specified as follows:

- Input: C $\in Fq^{q^2}$.
- Output: a $\in Fq^2$, that gives P(a) =C.

Given a received word c $\in Fq^{q^2}$ the codeword corresponding to P(X,Y), we want to recover the coordinates of a , a=(a1,..,ak) that gives P(a)=c .

Algorithm:

1. We have k coefficients that we want to know their values , so we will create an equation system with K equations.
   to get the k equations :
   - get all the monomials with degree <= d .
   - get the powers of x,y for each of the K-monomials as a tuple , assign it with powers list.
   - evaluate the monomials on each of the K-tuples in the power list (why? explanation of this follows the algorithm (*)), we will have a set of K-independent equations (that represents the evaluation of the monomials on the coordinates x and y in the power list). Now we have an equation system with K-variables (the coefficients of the polynomial) and we know for sure that it has one solution, that will give us the polynomial P(X,Y) that we are looking for.
   - To solve the equation system we represent it as follows:
     let M = the matrix KxK - the evaluation of the monomials on the coordinates x and y in the power list.
     let C = the coefficients vector - Kx1.
     let V = the values from codeword according to given x,y - Kx1
     The equation system:
     M x C = V .
2. To compute the vector C we must compute the matrix M^{-1} which is the inverse matrix of M , and then :

M^{-1} x M x C = M^{-1} x V // multiply two sides with M^{-1}

=> C = M^{-1} x V // M^{-1} x M = I

as we can see , that we can compute the C-vector by computing M^{-1} x v .
means that we can know the coefficients values , which is the coefficients of the original polynomial of the given code word .

(*) if we look at the monomials of degree at most d :

$x^i y^j$ such that $0 \le i + j \le d$

if we take a look at the polynomial of degree at most d :

$$P(x,y) = \sum_{i,j} c_{ij} x^i y^j \text{ such that } 0 \le i + j \le d$$

For each powers (x,y) if we apply the k-monomials on each power tuple we get k-independent equations. In Total we have K-variables and K-Equations and we know for sure that it has exactly one solution (because we know that the polynomial exists and its unique). Solving it will give us the original word (the coefficients).

Complexity:

- generating the monomials $x^i y^j$ such that $0 \le i + j \le d$, $i,j \in [0, q-1]$ = $O(q^2)$
- get the powers of each monomial = O(K) . (when K=(d+2 choose 2) #monomials)
- evaluate the monomials on the powers list = O(K * K)
- inverse matrix = we will assign it as $O(n^3)$
- Total = ($O(K^3)$ +O(K*K+ q^2)) = $O(K^3)$

A few important points related to Bivariate Reed-Muller codes :

- Because two distinct polynomials of degree at most d can agree on at most d/q -fraction of the points in Fq^2 , this code has distance δ= 1−d/q.
- Any polynomial of degree at most d is specified by one coefficient for each of the $(d+2 \; choose \; 2)$ monomials.
- The message length k=(d+2 choose 2) and n = q^2 , Thus the rate of this code is $\frac{(d+2 \; choose \; 2)}{q^2} \approx \frac{(1-\delta)^2}{2}$.

  Notice that this code cannot have rate more than ½.

6. Implementation:

A few points about my implementation :

• I wrote my code in Python 3.7.
• I used PyCharm as an IDE to write and run my code.
• I used some of the build-in libraries in Python.

Files:

rees_muller.py : contains all the code needed to implement the algorithm, it has 4 main classes.

Classes:

1. ReedMullerBV:

Defines a bi-variate reed muller code. This object can encode/correct/decode the code.

Fields:

q : represents the final field Fq.
d : the polynomial degree.

Functions:

- **encode:** Encodes the word by evaluating its Bivariate polynomial - P(X,Y) - at the coordinates x,y ∈ $[0, q-1]$. Then we get a code word of length q^2

  input: a word that represents the polynomial coefficients.
  output: the code word of length q^2

- **correct:** Corrects the value P(a) given the point a=(x,y).

  input: x: the x coordinate of the point we want to correct
  y: the y coordinate of the point we want to correct
  b: random point in Fq^2
  relative_code_word: used to get other points vals
  output: the corrected value of P(a)

- **decode:** Decodes a code word into the original word

  input: relative_code_word: a code word of size q^2 (a possibly corrupted)
  output: a coefficients list of the polynomial p that gives the relative_code_word

2. Monomial2D:

Defines a monomial with two variables.

Fields:

pow_x :the power of the variable x.

pow_y : the power of the variable y.

Functions:

- **eval:** returns the evaluations of the monomial on some x and y.
  input: values of x and y.
  output: the value of (x ^ pow_x) * (y ^ pow_y).

- **get_powers:** returns the monomial powers of x and y.
  input: takes the object (self) by default .
  output: an array contains the values pow_x and pow_y.

- **repr_:** prints a Monomial2D object.
  input: (self) , by default .
  output: string represents the object.

3. Poly2D:

Defines bi-variate polynomial, creates a polynomial object represented by monomials as Monomial2D objects. Given coefficients, d - polynomial degree and q - filed size.

Fields:

q : represents the final field Fq.
d : the polynomial degree.

Functions:

- **eval:** Evaluates the polynomial on the given x and y.
  input: values of x and y.
  output: the evaluation of p on x and y.

- **repr_:** prints a Poly2D object.
  input: (self) , by default .
  output: string represents the object.

4. Line2D:

Defines Linear line (a + b * t) , used in correcting algorithem .

Fields:

a : the point we want to compute its value - P(a).

b : a random point (x,y) in Fq^2.

Functions:

- **eval:** Evaluates the line given t value.
  input: t - an int value
  output: returns a vector v = (v0,v1) - the evaluation of the line a + b * t on the t input value, v0 = a0 + b0 * t and v1 = a1 + b1 * t .

- **repr_:** prints a Line2D object.
  input: (self) , by default .
  output: string represents the object.

Help functions:

is_prime_number : a functions used to check if the given q value is a prime number as it should be or not.

Help functions declared in the class ReedMuller2D ,used in decoding:

get_random_point : returns a random point of Fq^2 .
correct_point : Corrects the value P(a) given the point a=(x,y).
point_val_in_relative : Returns the value of the point (x,y) in the relative_code_word.
generate_monomials: returns all the monomials of Fq with max degree d .
get_codeword_according_to_indexes : returns the encoding of the given indexes x and y in the corrected_word.
monomials_eval_on_powlst: returns the evaluation of the monomial on a list of x-es and y-es.
pows_of_monomials : returns the pow_x , pow_y of the given monomial.

Libraries and build_in functions used in the implementation:

Libraries : random, scipy, numpy.
Functions used from the libraries :

from random import randint - to generate random point
from scipy.interpolate import lagrange - used to recover the correct P(a) in correct.
from numpy.linalg import inv - used to inverse a matrix in decoding the codeword.

The “main” Function:

This is the main function that runs all the algorithm of Reed-Muller Code:

- Creating a ReedMuller2D object - rm .

- Run the encoding function by using the class function given as input a list of coefficients - encode (rm.encode(coeffs))
- To correct some code word that is close to some original correct code word , we can correct one point value by using the class function correct , given the relative code word and the point we want to recover - (rm.correct(relative,point)) .
- To decode a code word back to list of coefficiens , we can use the class function decode, given the code word as input - (rm.decode(codeword)).

7. Running Examples :

Running Example 1

main function :

```
if __name__ == '__main__':

    rm = ReedMullerBV(q=3, d=1)
    code_word = rm.encode(word=[1, 2, 2])
    print("code_word : ", code_word)

    relative_code_word = code_word
    relative_code_word[1] = (code_word[1]+1) % rm.q
    print("relative_code_word : ", relative_code_word)

    corrected_word = rm.correct(relative_code_word, (0, 1))
    decoded_r = rm.decode(corrected_word)
    print("decoded word : ", decoded_r)
```

The Outputs of this inputs :

```
*** In encode Function ***
code_word :  [1, 0, 2, 0, 2, 1, 2, 1, 0]
relative_code_word :  [1, 1, 2, 0, 2, 1, 2, 1, 0]

*** In correct Function ***
  point a : [0, 1]
  point b : [1, 0]
  line = [0, 1]+ t * [1, 0]
t:  1
  point of (a + b * t) when t =  1  : [1, 1]
  - the evaluation of q on t : q(1) : 2
  - the point (t , q(t)) when t = 1 => (1, 2)
t:  2
  point of (a + b * t) when t =  2  : [2, 1]
  - the evaluation of q on t : q(2) : 1
  - the point (t , q(t)) when t = 2 => (2, 1)

Interpolations points : [(1, 2), (2, 1)]
  the x coordinates of the points : [1, 2]
  the y coordinates of the points : [2, 1]

Polynomial interpolation , gives us an approximation of the polynomial P(X,Y) :
-1 x + 3
 * q(0) = P(a) :  0.0

*** In decode Function ***

  generate monomials ->  monomials : [x^0y^0, x^0y^1, x^1y^0]
  indexes list (x's and y's) : [[0, 0], [0, 1], [1, 0]]
  values of monomials on indexes lists : [[1, 0, 0], [1, 1, 0], [1, 0, 1]]
  code word of (x,y)'s :  [1, 0.0, 0]
  inverse matrix :
 [[ 1.  0.  0.]
 [-1.  1.  0.]
 [-1.  0.  1.]]
  coeffections  :  [1. 2. 2.]

deoded word :  [1. 2. 2.]
```

Running Example 2

main Function :

```python
if __name__ == '__main__':

    rm = ReedMullerBV(q=5, d=2)
    code_word = rm.encode(word=[1, 2, 2, 3, 2, 1])
    print("code_word : ", code_word)

    relative_code_word = code_word
    relative_code_word[1] = (code_word[1]+1) % rm. q
    print("relative_code_word : ", relative_code_word)

    corrected_word = rm.correct(relative_code_word, (0, 1))
    decoded_r = rm.decode(corrected_word)
    print("decoded word : ", decoded_r)
```

The Outputs of this inputs :

```
*** In encode Function ***
code_word :  [1, 0, 3, 0, 1, 0, 1, 1, 0, 3, 1, 4, 1, 2, 2, 4, 4, 3, 1, 3, 4, 1, 2, 2, 1]
relative_code_word :  [1, 1, 3, 0, 1, 0, 1, 1, 0, 3, 1, 4, 1, 2, 2, 4, 4, 3, 1, 3, 4, 1, 2, 2, 1]

*** In correct Function ***
  point a : [0, 1]
  point b : [2, 0]
  line = [0, 1]+ t * [2, 0]
t:  1
  point of (a + b * t) when t =  1  : [2, 1]
  - the evaluation of q on t : q(1) : 4
  - the point (t , q(t)) when t = 1 => (1, 4)
t:  2
  point of (a + b * t) when t =  2  : [4, 1]
  - the evaluation of q on t : q(2) : 1
  - the point (t , q(t)) when t = 2 => (2, 1)
t:  3
  point of (a + b * t) when t =  3  : [1, 1]
  - the evaluation of q on t : q(3) : 1
  - the point (t , q(t)) when t = 3 => (3, 1)

Interpolations points : [(1, 4), (2, 1), (3, 1)]
  the x coordinates of the points : [1, 2, 3]
  the y coordinates of the points : [4, 1, 1]

Polynomial interpolation , gives us an approximation of the polynomial P(X,Y) :     2
1.5 x - 7.5 x + 10
 * q(0) = P(a) :  0.0

*** In decode Function ***

  generate monomials ->  monomials : [x^0y^0, x^0y^1, x^0y^2, x^1y^0, x^1y^1, x^2y^0]
  indexes list (x's and y's) : [[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [2, 0]]
  values of monomials on indexes lists :
 [[1, 0, 0, 0, 0, 0], [1, 1, 1, 0, 0, 0], [1, 2, 4, 0, 0, 0], [1, 0, 0, 1, 0, 1], [1, 1, 1, 1, 1, 1], [1, 0, 0, 2, 0, 4]]
  code word of (x,y)'s :  [1, 0.0, 3, 0, 1, 1]
  inverse matrix :
 [[ 1.   0.   0.   0.   0.   0. ]
 [-1.5  2.  -0.5  0.   0.   0. ]
 [ 0.5 -1.   0.5 -0.  -0.  -0. ]
 [-1.5  0.   0.   2.   0.  -0.5]
 [ 1.  -1.   0.  -1.   1.   0. ]
 [ 0.5 -0.  -0.  -1.  -0.   0.5]]
  coeffections  :  [1. 2. 2. 3. 2. 1.]

decoded word :  [1. 2. 2. 3. 2. 1.]
```