Introduction to Coding Theory

Daniel H. Luecking

MASC

November 8, 2023

What is coding theory

What is coding theory

Coding theory is about sending information as rapidly and accurately as possible. We will not cover ‘rapidly’, which deals with the compression of data, but only introduce some concepts related to accurately transmitting data.

What is coding theory

Coding theory is about sending information as rapidly and accurately as possible. We will not cover ‘rapidly’, which deals with the compression of data, but only introduce some concepts related to accurately transmitting data.

The theory recognizes that all transmission methods are imperfect and that, for one reason or another, a stream of bits may accumulate errors: a 0 becoming a 1 or the reverse.

What is coding theory

Coding theory is about sending information as rapidly and accurately as possible. We will not cover ‘rapidly’, which deals with the compression of data, but only introduce some concepts related to accurately transmitting data.

The theory recognizes that all transmission methods are imperfect and that, for one reason or another, a stream of bits may accumulate errors: a 0 becoming a 1 or the reverse. Detection of errors has been around since before computers: The ASCII-7 code was originally for sending messages to teletype machines.

What is coding theory

Coding theory is about sending information as rapidly and accurately as possible. We will not cover ‘rapidly’, which deals with the compression of data, but only introduce some concepts related to accurately transmitting data.

The theory recognizes that all transmission methods are imperfect and that, for one reason or another, a stream of bits may accumulate errors: a 0 becoming a 1 or the reverse. Detection of errors has been around since before computers: The ASCII-7 code was originally for sending messages to teletype machines. Seven bits were used to represent 128 different characters or functions. If even one bit was wrong the wrong letter would be printed or a newline or pagebreak could appear in the middle of a word.

What is coding theory

Coding theory is about sending information as rapidly and accurately as possible. We will not cover ‘rapidly’, which deals with the compression of data, but only introduce some concepts related to accurately transmitting data.

The theory recognizes that all transmission methods are imperfect and that, for one reason or another, a stream of bits may accumulate errors: a 0 becoming a 1 or the reverse. Detection of errors has been around since before computers: The ASCII-7 code was originally for sending messages to teletype machines. Seven bits were used to represent 128 different characters or functions. If even one bit was wrong the wrong letter would be printed or a newline or pagebreak could appear in the middle of a word.

To detect these errors ‘parity schemes’ were used. Instead of 7 bits per character, 8 bits could be sent, with an extra bit added to make the total number of 1’s in the string even.

Even-parity error detection

For example, to code an ‘A’, the ASCII code has two 1’s so we would add a 0 (typically at the left end, as in the example below) and transmit that 8-bit string.

Even-parity error detection

For example, to code an ‘A’, the ASCII code has two 1’s so we would add a 0 (typically at the left end, as in the example below) and transmit that 8-bit string. While a ‘C’ has three 1’s so we would add a 1.

Even-parity error detection

For example, to code an 'A', the ASCII code has two 1's so we would add a 0 (typically at the left end, as in the example below) and transmit that 8-bit string. While a 'C' has three 1's so we would add a 1. That is,

$$A \xrightarrow{\text{ASCII}} (100\,0001) \xrightarrow{\text{parity}} (0100\,0001)$$
$$C \xrightarrow{\text{ASCII}} (100\,0011) \xrightarrow{\text{parity}} (1100\,0011)$$

In both cases the new code has an even number of 1s.

Even-parity error detection

For example, to code an 'A', the ASCII code has two 1's so we would add a 0 (typically at the left end, as in the example below) and transmit that 8-bit string. While a 'C' has three 1's so we would add a 1. That is,

$$A \xrightarrow{\text{ASCII}} (100\,0001) \xrightarrow{\text{parity}} (0100\,0001)$$
$$C \xrightarrow{\text{ASCII}} (100\,0011) \xrightarrow{\text{parity}} (1100\,0011)$$

In both cases the new code has an even number of 1s. If an 8-bit string arrives with an odd number of 1s we know that an error has occurred in at least one bit.

Even-parity error detection

For example, to code an ‘A’, the ASCII code has two 1’s so we would add a 0 (typically at the left end, as in the example below) and transmit that 8-bit string. While a ‘C’ has three 1’s so we would add a 1. That is,

$$A \xrightarrow{\text{ASCII}} (100\,0001) \xrightarrow{\text{parity}} (0100\,0001)$$
$$C \xrightarrow{\text{ASCII}} (100\,0011) \xrightarrow{\text{parity}} (1100\,0011)$$

In both cases the new code has an even number of 1s. If an 8-bit string arrives with an odd number of 1s we know that an error has occurred in at least one bit. We don’t know which bit has changed, but we know not to trust the data and perhaps can ask for it to be sent again.

Even-parity error detection

For example, to code an 'A', the ASCII code has two 1's so we would add a 0 (typically at the left end, as in the example below) and transmit that 8-bit string. While a 'C' has three 1's so we would add a 1. That is,

$$A \xrightarrow{\text{ASCII}} (100\,0001) \xrightarrow{\text{parity}} (0100\,0001)$$
$$C \xrightarrow{\text{ASCII}} (100\,0011) \xrightarrow{\text{parity}} (1100\,0011)$$

In both cases the new code has an even number of 1s. If an 8-bit string arrives with an odd number of 1s we know that an error has occurred in at least one bit. We don't know which bit has changed, but we know not to trust the data and perhaps can ask for it to be sent again.

An example of an error-correction code is the following: send each bit three times.

Even-parity error detection

For example, to code an 'A', the ASCII code has two 1's so we would add a 0 (typically at the left end, as in the example below) and transmit that 8-bit string. While a 'C' has three 1's so we would add a 1. That is,

$$A \xrightarrow{\text{ASCII}} (100\,0001) \xrightarrow{\text{parity}} (0100\,0001)$$
$$C \xrightarrow{\text{ASCII}} (100\,0011) \xrightarrow{\text{parity}} (1100\,0011)$$

In both cases the new code has an even number of 1s. If an 8-bit string arrives with an odd number of 1s we know that an error has occurred in at least one bit. We don't know which bit has changed, but we know not to trust the data and perhaps can ask for it to be sent again.

An example of an error-correction code is the following: send each bit three times. Instead of sending the actual data like $1001\ldots$, send $111\,000\,000\,111\ldots$. If a single bit is changed (say a 111 becomes 101) we know not only that there is an error, but where it is.

The probability of error

Adding extra bits lengthens a message and so actually adds more opportunities for errors.

The probability of error

Adding extra bits lengthens a message and so actually adds more opportunities for errors. We should be sure not to lengthen the message so much as to *increase* the likelihood of error beyond our ability to correct it..

The probability of error

Adding extra bits lengthens a message and so actually adds more opportunities for errors. We should be sure not to lengthen the message so much as to *increase* the likelihood of error beyond our ability to correct it..

In the following calculations p is the probability of error for 1 bit.

The probability of error

Adding extra bits lengthens a message and so actually adds more opportunities for errors. We should be sure not to lengthen the message so much as to *increase* the likelihood of error beyond our ability to correct it..

In the following calculations p is the probability of error for 1 bit. We assume that

1. $p < 1/2$,

The probability of error

Adding extra bits lengthens a message and so actually adds more opportunities for errors. We should be sure not to lengthen the message so much as to *increase* the likelihood of error beyond our ability to correct it..

In the following calculations p is the probability of error for 1 bit. We assume that

1. $p < 1/2$,
2. the likelihood of a change in a bit does not depend on whether it is a 0 or a 1,

The probability of error

Adding extra bits lengthens a message and so actually adds more opportunities for errors. We should be sure not to lengthen the message so much as to *increase* the likelihood of error beyond our ability to correct it..

In the following calculations p is the probability of error for 1 bit. We assume that

1. $p < 1/2$,
2. the likelihood of a change in a bit does not depend on whether it is a 0 or a 1,
3. the likelihood of a change in a bit does not depend on what happens to other bits around it.

Example: suppose we want to send 3 bits of information, and appending another 3 bits gives us the ability to correct any one bit of the 6. Is this better?

The probability of error

Adding extra bits lengthens a message and so actually adds more opportunities for errors. We should be sure not to lengthen the message so much as to *increase* the likelihood of error beyond our ability to correct it..

In the following calculations p is the probability of error for 1 bit. We assume that

1. $p < 1/2$,
2. the likelihood of a change in a bit does not depend on whether it is a 0 or a 1,
3. the likelihood of a change in a bit does not depend on what happens to other bits around it.

Example: suppose we want to send 3 bits of information, and appending another 3 bits gives us the ability to correct any one bit of the 6. Is this better?

If we send only the 3 bits, the probability that all bits are correct is $(1-p)^3$.

Basic principles

If we send 6 bits, with the ability to correct any 1-bit error, the correct message gets through if there are no errors (probability $(1-p)^6$) or one error (probability $6p(1-p)^5$).

Basic principles

If we send 6 bits, with the ability to correct any 1-bit error, the correct message gets through if there are no errors (probability $(1-p)^6$) or one error (probability $6p(1-p)^5$). So we need $(1-p)^6 + 6p(1-p)^5 > (1-p)^3$, which is true if $p \leq 0.44$.

Basic principles

If we send 6 bits, with the ability to correct any 1-bit error, the correct message gets through if there are no errors (probability $(1-p)^6$) or one error (probability $6p(1-p)^5$). So we need $(1-p)^6 + 6p(1-p)^5 > (1-p)^3$, which is true if $p \leq 0.44$.

In more realistic problems we send much larger chunks of information and require much more error correcting ability, but there is a basic theorem that for any $p < 1/2$ there is some error-correcting scheme that approaches perfect reliability.

Basic principles

If we send 6 bits, with the ability to correct any 1-bit error, the correct message gets through if there are no errors (probability $(1-p)^6$) or one error (probability $6p(1-p)^5$). So we need $(1-p)^6 + 6p(1-p)^5 > (1-p)^3$, which is true if $p \leq 0.44$.

In more realistic problems we send much larger chunks of information and require much more error correcting ability, but there is a basic theorem that for any $p < 1/2$ there is some error-correcting scheme that approaches perfect reliability.

All error correction schemes have the following set-up. Messages are broken into parts which are strings of bits with a chosen length m, that is, elements of $\mathbb{Z}_2^m$; these are called 'words'.

Basic principles

If we send 6 bits, with the ability to correct any 1-bit error, the correct message gets through if there are no errors (probability $(1-p)^6$) or one error (probability $6p(1-p)^5$). So we need $(1-p)^6 + 6p(1-p)^5 > (1-p)^3$, which is true if $p \leq 0.44$.

In more realistic problems we send much larger chunks of information and require much more error correcting ability, but there is a basic theorem that for any $p < 1/2$ there is some error-correcting scheme that approaches perfect reliability.

All error correction schemes have the following set-up. Messages are broken into parts which are strings of bits with a chosen length m, that is, elements of $\mathbb{Z}_2^m$; these are called 'words'. Then extra bits are appended to a message word w to produce a longer 'code word' c in $\mathbb{Z}_2^n$ with $n > m$.

Basic principles

If we send 6 bits, with the ability to correct any 1-bit error, the correct message gets through if there are no errors (probability $(1-p)^6$) or one error (probability $6p(1-p)^5$). So we need $(1-p)^6 + 6p(1-p)^5 > (1-p)^3$, which is true if $p \le 0.44$.

In more realistic problems we send much larger chunks of information and require much more error correcting ability, but there is a basic theorem that for any $p < 1/2$ there is some error-correcting scheme that approaches perfect reliability.

All error correction schemes have the following set-up. Messages are broken into parts which are strings of bits with a chosen length m, that is, elements of $\mathbb{Z}_2^m$; these are called 'words'. Then extra bits are appended to a message word w to produce a longer 'code word' c in $\mathbb{Z}_2^n$ with $n > m$.

The process for doing this is called 'encoding'. The code word c is sent over a communication channel with possible errors introduced. Call the result r (for 'received word').

Then r is tested for errors and possibly corrected to get (hopefully) c again.

Then r is tested for errors and possibly corrected to get (hopefully) c again.

Finally, the extra bits are removed from c (the decoding step) to obtain the original message word w:

Then r is tested for errors and possibly corrected to get (hopefully) c again.

Finally, the extra bits are removed from c (the decoding step) to obtain the original message word w:

$$\begin{array}{ccccccccc} w & \xrightarrow{\text{encode}} & c & \xrightarrow{\text{send}} & r & \xrightarrow{\text{correct}} & c & \xrightarrow{\text{decode}} & w \\ \mathbb{Z}_2^m & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^m \end{array}$$

Then r is tested for errors and possibly corrected to get (hopefully) c again.

Finally, the extra bits are removed from c (the decoding step) to obtain the original message word w:

$$\begin{array}{ccccccccc} w & \xrightarrow{\text{encode}} & c & \xrightarrow{\text{send}} & r & \xrightarrow{\text{correct}} & c & \xrightarrow{\text{decode}} & w \\ \mathbb{Z}_2^m & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^m \end{array}$$

The basic principle for encoding is that the added bits in the encoding process are obtained in a computable way which we might represent by $c = E(w)$ (E for 'encoding').

Then r is tested for errors and possibly corrected to get (hopefully) c again.

Finally, the extra bits are removed from c (the decoding step) to obtain the original message word w:

$$\begin{array}{ccccccccc} w & \xrightarrow{\text{encode}} & c & \xrightarrow{\text{send}} & r & \xrightarrow{\text{correct}} & c & \xrightarrow{\text{decode}} & w \\ \mathbb{Z}_2^m & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^m \end{array}$$

The basic principle for encoding is that the added bits in the encoding process are obtained in a computable way which we might represent by $c = E(w)$ (E for 'encoding'). These are called 'parity bits' in analogy to the ASCII example.

Then r is tested for errors and possibly corrected to get (hopefully) c again.

Finally, the extra bits are removed from c (the decoding step) to obtain the original message word w:

$$\begin{array}{ccccccccc} w & \xrightarrow{\text{encode}} & c & \xrightarrow{\text{send}} & r & \xrightarrow{\text{correct}} & c & \xrightarrow{\text{decode}} & w \\ \mathbb{Z}_2^m & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^m \end{array}$$

The basic principle for encoding is that the added bits in the encoding process are obtained in a computable way which we might represent by $c = E(w)$ (E for 'encoding'). These are called 'parity bits' in analogy to the ASCII example.

Then the set of all possible $E(w)$ as w ranges through $\mathbb{Z}_2^m$ is called the 'code'.

Then r is tested for errors and possibly corrected to get (hopefully) c again.

Finally, the extra bits are removed from c (the decoding step) to obtain the original message word w:

$$\begin{array}{ccccccccc} w & \xrightarrow{\text{encode}} & c & \xrightarrow{\text{send}} & r & \xrightarrow{\text{correct}} & c & \xrightarrow{\text{decode}} & w \\ \mathbb{Z}_2^m & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^n & \longrightarrow & \mathbb{Z}_2^m \end{array}$$

The basic principle for encoding is that the added bits in the encoding process are obtained in a computable way which we might represent by $c = E(w)$ (E for 'encoding'). These are called 'parity bits' in analogy to the ASCII example.

Then the set of all possible $E(w)$ as w ranges through $\mathbb{Z}_2^m$ is called the 'code'. So, a 'code' is just a set $\mathcal{C}$ of strings in $\mathbb{Z}_2^n$. It is not all of $\mathbb{Z}_2^n$ because $\mathcal{C}$ has only 2^m elements, one for each w in $\mathbb{Z}_2^m$, while $\mathbb{Z}_2^n$ has 2^n elements and $n > m$.

Error detection works simply by checking whether the received word r belongs to $\mathcal{C}$. If r is not in $\mathcal{C}$ we know there was an error. If it is in $\mathcal{C}$ we assume no error.

Error detection works simply by checking whether the received word r belongs to $\mathcal{C}$. If r is not in $\mathcal{C}$ we know there was an error. If it is in $\mathcal{C}$ we assume no error. This assumption is valid only if we have designed $\mathcal{C}$ so that the probability of an error changing one code word into another code word is smaller that the probability of no error.

Error detection works simply by checking whether the received word r belongs to $\mathcal{C}$. If r is not in $\mathcal{C}$ we know there was an error. If it is in $\mathcal{C}$ we assume no error. This assumption is valid only if we have designed $\mathcal{C}$ so that the probability of an error changing one code word into another code word is smaller that the probability of no error.

A simple error-correction example

Error detection works simply by checking whether the received word r belongs to $\mathcal{C}$. If r is not in $\mathcal{C}$ we know there was an error. If it is in $\mathcal{C}$ we assume no error. This assumption is valid only if we have designed $\mathcal{C}$ so that the probability of an error changing one code word into another code word is smaller that the probability of no error.

A simple error-correction example

A good code $\mathcal{C}$ is designed so that every pair of words differ in a significant number of places. That is, the probability of a code word becoming a different code word must be small.

Error detection works simply by checking whether the received word r belongs to $\mathcal{C}$. If r is not in $\mathcal{C}$ we know there was an error. If it is in $\mathcal{C}$ we assume no error. This assumption is valid only if we have designed $\mathcal{C}$ so that the probability of an error changing one code word into another code word is smaller that the probability of no error.

A simple error-correction example

A good code $\mathcal{C}$ is designed so that every pair of words differ in a significant number of places. That is, the probability of a code word becoming a different code word must be small.

The following code was produced by adding 3 bits to words in $\mathbb{Z}_2^3$ to produce strings in $\mathbb{Z}_2^6$:

Error detection works simply by checking whether the received word r belongs to $\mathcal{C}$. If r is not in $\mathcal{C}$ we know there was an error. If it is in $\mathcal{C}$ we assume no error. This assumption is valid only if we have designed $\mathcal{C}$ so that the probability of an error changing one code word into another code word is smaller that the probability of no error.

A simple error-correction example

A good code $\mathcal{C}$ is designed so that every pair of words differ in a significant number of places. That is, the probability of a code word becoming a different code word must be small.

The following code was produced by adding 3 bits to words in $\mathbb{Z}_2^3$ to produce strings in $\mathbb{Z}_2^6$:

$$\mathcal{C} = \{(000\,000), (001\,011), (010\,110), (100\,101),\\ (011\,101), (101\,110), (110\,011), (111\,000)\}$$

In this example it takes at least 3 errors to turn any one of these into another one.

In this example it takes at least 3 errors to turn any one of these into another one. Assuming this, we have

1. Any changes to 1 or 2 bits is detectable as an error. (We say $\mathcal{C}$ has 2-bit *error detection*.)

In this example it takes at least 3 errors to turn any one of these into another one. Assuming this, we have

1. Any changes to 1 or 2 bits is detectable as an error. (We say $\mathcal{C}$ has 2-bit *error detection*.)
2. An error in only one bit can be corrected. (We say $\mathcal{C}$ has 1-bit *error correction*.)

In this example it takes at least 3 errors to turn any one of these into another one. Assuming this, we have

1. Any changes to 1 or 2 bits is detectable as an error. (We say $\mathcal{C}$ has 2-bit *error detection*.)
2. An error in only one bit can be corrected. (We say $\mathcal{C}$ has 1-bit *error correction*.)

This relies on multiple errors being less likely than one error. This is true under conditions like one of the previous slides, and depends on p.

In this example it takes at least 3 errors to turn any one of these into another one. Assuming this, we have

1. Any changes to 1 or 2 bits is detectable as an error. (We say $\mathcal{C}$ has 2-bit *error detection*.)
2. An error in only one bit can be corrected. (We say $\mathcal{C}$ has 1-bit *error correction*.)

This relies on multiple errors being less likely than one error. This is true under conditions like one of the previous slides, and depends on p.

Definition

The *Hamming distance* $d(w, w')$ *between* w *and* w' if the number of bits that need to be changed to turn the word w into the word w'.

Suppose two code words c and c' have $d(c, c') = j$ for some positive integer j.

Suppose two code words c and c' have $d(c, c') = j$ for some positive integer j. If r is received and $d(c, r) < j/2$ then also $d(c', r) > j/2$, and so r is more likely to have been sent as c than as c'.

Suppose two code words c and c' have $d(c, c') = j$ for some positive integer j. If r is received and $d(c, r) < j/2$ then also $d(c', r) > j/2$, and so r is more likely to have been sent as c than as c'. (Fewer errors are more likely if p is small).

Suppose two code words c and c' have $d(c, c') = j$ for some positive integer j. If r is received and $d(c, r) < j/2$ then also $d(c', r) > j/2$, and so r is more likely to have been sent as c than as c'. (Fewer errors are more likely if p is small).

A key parameter for any code $\mathcal{C}$ is the minimum distance between code words, which we call d.

Suppose two code words c and c' have $d(c, c') = j$ for some positive integer j. If r is received and $d(c, r) < j/2$ then also $d(c', r) > j/2$, and so r is more likely to have been sent as c than as c'. (Fewer errors are more likely if p is small).

A key parameter for any code $\mathcal{C}$ is the minimum distance between code words, which we call d. It has the property that any error in k bits can be *detected* if $k < d$ and can be *corrected* if $2k < d$.

Suppose two code words c and c' have $d(c, c') = j$ for some positive integer j. If r is received and $d(c, r) < j/2$ then also $d(c', r) > j/2$, and so r is more likely to have been sent as c than as c'. (Fewer errors are more likely if p is small).

A key parameter for any code $\mathcal{C}$ is the minimum distance between code words, which we call d. It has the property that any error in k bits can be *detected* if $k < d$ and can be *corrected* if $2k < d$.

For example, if the minimum distance d is 6, then any errors in 5 bits or fewer are detectable and any errors in 2 bits or fewer is correctable.

Suppose two code words c and c' have $d(c, c') = j$ for some positive integer j. If r is received and $d(c, r) < j/2$ then also $d(c', r) > j/2$, and so r is more likely to have been sent as c than as c'. (Fewer errors are more likely if p is small).

A key parameter for any code $\mathcal{C}$ is the minimum distance between code words, which we call d. It has the property that any error in k bits can be *detected* if $k < d$ and can be *corrected* if $2k < d$.

For example, if the minimum distance d is 6, then any errors in 5 bits or fewer are detectable and any errors in 2 bits or fewer is correctable. But 3-bit errors could land "half-way between" code words and correcting to the nearest one is impossible.

Suppose two code words c and c' have $d(c, c') = j$ for some positive integer j. If r is received and $d(c, r) < j/2$ then also $d(c', r) > j/2$, and so r is more likely to have been sent as c than as c'. (Fewer errors are more likely if p is small).

A key parameter for any code $\mathcal{C}$ is the minimum distance between code words, which we call d. It has the property that any error in k bits can be *detected* if $k < d$ and can be *corrected* if $2k < d$.

For example, if the minimum distance d is 6, then any errors in 5 bits or fewer are detectable and any errors in 2 bits or fewer is correctable. But 3-bit errors could land "half-way between" code words and correcting to the nearest one is impossible.

Our previous 6-bit example has $d = 3$ which has 2-bit detectability and 1-bit correctability.

How does group theory get involved? It does so because $\mathbb{Z}_2^n$ is a group under bit-wise addition mod 2, and an error can be viewed as addition.

How does group theory get involved? It does so because $\mathbb{Z}_2^n$ is a group under bit-wise addition mod 2, and an error can be viewed as addition.

For example, suppose $c = (101\,110)$ is a code word and $e = (001\,000)$. Then $c + e = (100\,110)$, which is c changed in third position from a 1 to a $0 = 1 + 1 \bmod 2$.

How does group theory get involved? It does so because $\mathbb{Z}_2^n$ is a group under bit-wise addition mod 2, and an error can be viewed as addition.

For example, suppose $c = (101\,110)$ is a code word and $e = (001\,000)$. Then $c + e = (100\,110)$, which is c changed in third position from a 1 to a $0 = 1 + 1 \bmod 2$. Also $c + (010\,000)$ changes the 0 in second position to 1.

How does group theory get involved? It does so because $\mathbb{Z}_2^n$ is a group under bit-wise addition mod 2, and an error can be viewed as addition.

For example, suppose $c = (101\,110)$ is a code word and $e = (001\,000)$. Then $c + e = (100\,110)$, which is c changed in third position from a 1 to a $0 = 1 + 1 \bmod 2$. Also $c + (010\,000)$ changes the 0 in second position to 1.

If e has more than a single 1, that would correspond to more errors. We use this concept in conjunction with a formula for the Hamming distance.

How does group theory get involved? It does so because $\mathbb{Z}_2^n$ is a group under bit-wise addition mod 2, and an error can be viewed as addition.

For example, suppose $c = (101\,110)$ is a code word and $e = (001\,000)$. Then $c + e = (100\,110)$, which is c changed in third position from a 1 to a $0 = 1 + 1 \bmod 2$. Also $c + (010\,000)$ changes the 0 in second position to 1.

If e has more than a single 1, that would correspond to more errors. We use this concept in conjunction with a formula for the Hamming distance.

Definition

The *weight* of a word is the number of 1s in its string.

How does group theory get involved? It does so because $\mathbb{Z}_2^n$ is a group under bit-wise addition mod 2, and an error can be viewed as addition.

For example, suppose $c = (101\,110)$ is a code word and $e = (001\,000)$. Then $c + e = (100\,110)$, which is c changed in third position from a 1 to a $0 = 1 + 1 \bmod 2$. Also $c + (010\,000)$ changes the 0 in second position to 1.

If e has more than a single 1, that would correspond to more errors. We use this concept in conjunction with a formula for the Hamming distance.

Definition

The *weight* of a word is the number of 1s in its string.

For example the weight of the above $c = (101\,110)$ is 4. We use $\mathrm{wt}(w)$ for the weight of w.

How does group theory get involved? It does so because $\mathbb{Z}_2^n$ is a group under bit-wise addition mod 2, and an error can be viewed as addition.

For example, suppose $c = (101\,110)$ is a code word and $e = (001\,000)$. Then $c + e = (100\,110)$, which is c changed in third position from a 1 to a $0 = 1 + 1 \bmod 2$. Also $c + (010\,000)$ changes the 0 in second position to 1.

If e has more than a single 1, that would correspond to more errors. We use this concept in conjunction with a formula for the Hamming distance.

Definition

The *weight* of a word is the number of 1s in its string.

For example the weight of the above $c = (101\,110)$ is 4. We use $\mathrm{wt}(w)$ for the weight of w.

Note that if we add w and w', then $w + w'$ has a 1 in those positions where w and w' are different and a 0 in positions where they are the same.

Thus the distance $d(w, w')$ is the number of places $w + w'$ has a 1:

$$d(w, w') = \mathrm{wt}(w + w') \quad \text{(bitwise addition mod 2)}$$

Thus the distance $d(w, w')$ is the number of places $w + w'$ has a 1:

$$d(w, w') = \mathrm{wt}(w + w') \quad \text{(bitwise addition mod 2)}$$

Definition

If a code $\mathcal{C}$ is a subgroup of $\mathbb{Z}_2^n$ we call it a *group code*.

Thus the distance $d(w, w')$ is the number of places $w + w'$ has a 1:

$$d(w, w') = \mathrm{wt}(w + w') \quad \text{(bitwise addition mod 2)}$$

Definition

If a code $\mathcal{C}$ is a subgroup of $\mathbb{Z}_2^n$ we call it a *group code*.

Since a group code must be closed under addition, $d(c, c') = \mathrm{wt}(c + c')$ is the weight of another code word.

Thus the distance $d(w, w')$ is the number of places $w + w'$ has a 1:

$$d(w, w') = \mathrm{wt}(w + w') \text{ (bitwise addition mod 2)}$$

Definition

If a code $\mathcal{C}$ is a subgroup of $\mathbb{Z}_2^n$ we call it a *group code*.

Since a group code must be closed under addition, $d(c, c') = \mathrm{wt}(c + c')$ is the weight of another code word. So, we can determine the minimum distance between code words by determining the weight of all words in $\mathcal{C}$ (except the zero string) and taking the smallest.

Thus the distance $d(w, w')$ is the number of places $w + w'$ has a 1:

$$d(w, w') = \mathrm{wt}(w + w') \text{ (bitwise addition mod 2)}$$

Definition

If a code $\mathcal{C}$ is a subgroup of $\mathbb{Z}_2^n$ we call it a *group code*.

Since a group code must be closed under addition, $d(c, c') = \mathrm{wt}(c + c')$ is the weight of another code word. So, we can determine the minimum distance between code words by determining the weight of all words in $\mathcal{C}$ (except the zero string) and taking the smallest.

Theorem

For a group code, the minimum distance between code words equals the minimum weight of the nonzero code words.

Take the earlier example:

$$\mathcal{C} = \{(000\,000), (001\,011), (010\,110), (100\,101),$$
$$(011\,101), (101\,110), (110\,011), (111\,000)\}$$

Take the earlier example:

$$\mathcal{C} = \{(000\,000), (001\,011), (010\,110), (100\,101), \\ (011\,101), (101\,110), (110\,011), (111\,000)\}$$

I created this to be a group code so, since we get the following 7 weights for the nonzero strings: $3, 3, 3, 4, 4, 4, 3$, the minimum distance is $d = 3$.

Take the earlier example:

$$\begin{aligned}\mathcal{C} = \{&(000\,000), (001\,011), (010\,110), (100\,101),\\ &(011\,101), (101\,110), (110\,011), (111\,000)\}\end{aligned}$$

I created this to be a group code so, since we get the following 7 weights for the nonzero strings: $3, 3, 3, 4, 4, 4, 3$, the minimum distance is $d = 3$.

This means we have 2-bit error detection and 1-bit error correction.

Take the earlier example:

$$\mathcal{C} = \{(000\,000), (001\,011), (010\,110), (100\,101), \\ (011\,101), (101\,110), (110\,011), (111\,000)\}$$

I created this to be a group code so, since we get the following 7 weights for the nonzero strings: $3, 3, 3, 4, 4, 4, 3$, the minimum distance is $d = 3$.

This means we have 2-bit error detection and 1-bit error correction.

A more important advantage of a group codes is this: determining if r is in the code $\mathcal{C}$, is an easy (for a computer) calculation that both detects and corrects errors (when possible).

Take the earlier example:

$$\mathcal{C} = \{(000\,000), (001\,011), (010\,110), (100\,101),\\ (011\,101), (101\,110), (110\,011), (111\,000)\}$$

I created this to be a group code so, since we get the following 7 weights for the nonzero strings: $3, 3, 3, 4, 4, 4, 3$, the minimum distance is $d = 3$.

This means we have 2-bit error detection and 1-bit error correction.

A more important advantage of a group codes is this: determining if r is in the code $\mathcal{C}$, is an easy (for a computer) calculation that both detects and corrects errors (when possible). We don't have to do an exhaustive search of $\mathcal{C}$ (impossible if $\mathcal{C}$ has 2^m elements with say $m > 100$).

Practical considerations

In typical applications we want to send significantly sized code words, for example $n = 256$ or higher.

Take the earlier example:

$$\begin{aligned}\mathcal{C} = \{&(000\,000), (001\,011), (010\,110), (100\,101),\\ &(011\,101), (101\,110), (110\,011), (111\,000)\}\end{aligned}$$

I created this to be a group code so, since we get the following 7 weights for the nonzero strings: $3, 3, 3, 4, 4, 4, 3$, the minimum distance is $d = 3$.

This means we have 2-bit error detection and 1-bit error correction.

A more important advantage of a group codes is this: determining if r is in the code $\mathcal{C}$, is an easy (for a computer) calculation that both detects and corrects errors (when possible). We don't have to do an exhaustive search of $\mathcal{C}$ (impossible if $\mathcal{C}$ has 2^m elements with say $m > 100$).

Practical considerations

In typical applications we want to send significantly sized code words, for example $n = 256$ or higher. Our earlier statements like "more errors are less likely than fewer errors" are only true if p is less than around $1/n$.

Take the earlier example:

$$\mathcal{C} = \{(000\,000), (001\,011), (010\,110), (100\,101),\\ (011\,101), (101\,110), (110\,011), (111\,000)\}$$

I created this to be a group code so, since we get the following 7 weights for the nonzero strings: $3, 3, 3, 4, 4, 4, 3$, the minimum distance is $d = 3$.

This means we have 2-bit error detection and 1-bit error correction.

A more important advantage of a group codes is this: determining if r is in the code $\mathcal{C}$, is an easy (for a computer) calculation that both detects and corrects errors (when possible). We don't have to do an exhaustive search of $\mathcal{C}$ (impossible if $\mathcal{C}$ has 2^m elements with say $m > 100$).

Practical considerations

In typical applications we want to send significantly sized code words, for example $n = 256$ or higher. Our earlier statements like "more errors are less likely than fewer errors" are only true if p is less than around $1/n$. This might typically be true, but transmission methods may have to monitor the reliability of the communication channel and estimate p in real time.

Some of the probability formulas I used earlier assumed that an error in a bit does not depend on what happens in other bits.

Some of the probability formulas I used earlier assumed that an error in a bit does not depend on what happens in other bits. This is the most likely of the assumptions to not be true in real life.

Some of the probability formulas I used earlier assumed that an error in a bit does not depend on what happens in other bits. This is the most likely of the assumptions to not be true in real life.

Errors caused by external events (e.g., voltage spikes) are typically "burst errors": several bits in a row may become essentially random, so an error in one bit makes it more likely that nearby bits are wrong.

Some of the probability formulas I used earlier assumed that an error in a bit does not depend on what happens in other bits. This is the most likely of the assumptions to not be true in real life.

Errors caused by external events (e.g., voltage spikes) are typically "burst errors": several bits in a row may become essentially random, so an error in one bit makes it more likely that nearby bits are wrong.

This doesn't have to affect the probability analysis: the encoding method can include scrambling the bits, then close-together errors become far apart errors when the message is unscrambled.

Some of the probability formulas I used earlier assumed that an error in a bit does not depend on what happens in other bits. This is the most likely of the assumptions to not be true in real life.

Errors caused by external events (e.g., voltage spikes) are typically "burst errors": several bits in a row may become essentially random, so an error in one bit makes it more likely that nearby bits are wrong.

This doesn't have to affect the probability analysis: the encoding method can include scrambling the bits, then close-together errors become far apart errors when the message is unscrambled.

Modern error correction codes use methods that model messages not as elements of $\mathbb{Z}_2^m$ but as elements of $F_{2^k}^m$ where F_{2^k} is a ring with 2^k elements that has special properties.

Some of the probability formulas I used earlier assumed that an error in a bit does not depend on what happens in other bits. This is the most likely of the assumptions to not be true in real life.

Errors caused by external events (e.g., voltage spikes) are typically "burst errors": several bits in a row may become essentially random, so an error in one bit makes it more likely that nearby bits are wrong.

This doesn't have to affect the probability analysis: the encoding method can include scrambling the bits, then close-together errors become far apart errors when the message is unscrambled.

Modern error correction codes use methods that model messages not as elements of $\mathbb{Z}_2^m$ but as elements of $F_{2^k}^m$ where F_{2^k} is a ring with 2^k elements that has special properties.

In these schemes, any errors within k consecutive bits are as correctable as a one bit error. This makes burst errors manageable.