Datenkompression

Einfache Algorithmen

Digitale AV Technik, MIB 5

Aus Sicht der Informationstheorie

Problem	**Kompression**	**Fehlerkorrektur**
Ziel	Effizienz	Verlässlichkeit
Anwendung	**Quellencodierung**	Kanalcodierung

Algorithmische Perspektive

Problem	**Kompression**
Algorithmen	**Shannon–Fano coding**
	Huffman-Code
	Arithmetische Codierung, CABAC
	Lempel-Ziv-Welch

Quellencodierung nach Shannon-Fano

- Shannon and Fano haben leicht unterschiedliche Beschreibungen veröffentlicht aber auch zusammen gearbeitet.
- Beide Versionen führen zu einem ähnlichen Ergebnis

Grundprinzipien der Shannon-Fano-Codierung

- Wahrscheinlichkeitsbasierte Kodierung: Kodiert Symbole basierend auf ihrer Auftrittswahrscheinlichkeit.
- Binärbaumstruktur: Nutzt einen Binärbaum und teilt Symbolgruppen gleichmäßig.
- Präfixfreier Code: Kein Code ist ein Präfix eines anderen, was Dekodierfehler verhindert.

Fano-Codierung: Schritte zur Implementierung - Schritt 1

Liste der Symbole und Häufigkeiten entwickeln

- Für eine gegebene Liste von Symbolen eine entsprechende Liste von Wahrscheinlichkeiten oder Häufigkeitswerten erstellen.
- Dadurch wird die relative Auftretenshäufigkeit jedes Symbols bestimmt.

Fano-Codierung: Schritte zur Implementierung - Schritt 2

Symbole nach Häufigkeit sortieren

- Die Symbol-Liste nach Häufigkeit sortieren.
- Häufig auftretende Symbole links anordnen, seltenere Symbole rechts.

Fano-Codierung: Schritte zur Implementierung - Schritt 3

Liste in zwei Teile aufteilen

- Die Liste in zwei Teile aufteilen, sodass die Summe der Häufigkeitswerte des linken Teils der Summe des rechten Teils möglichst nahekommt.

Fano-Codierung: Schritte zur Implementierung - Schritt 4

Binäre Zuweisung

- Der linke Teil der Liste erhält das Binärzeichen `0`, der rechte Teil das Binärzeichen `1`.
- Die Codes der Symbole im linken Teil beginnen also alle mit `0`, und die Codes im rechten Teil beginnen alle mit `1`.

Fano-Codierung: Schritte zur Implementierung - Schritt 5

Rekursive Anwendung

- Die Schritte 3 und 4 rekursiv auf die beiden Hälften anwenden.
- Gruppen weiter unterteilen und Bits zu den Codes hinzufügen, bis jedes Symbol ein entsprechendes Blatt im Baum wird.

Fano-Codierung: Implementierung - Schritt 1

Liste der Symbole und Häufigkeiten entwickeln

```
def get_character_counts(text):
    # simply use the Counter class from the collections module
    # to count the occurrences of each character
    return Counter(text)
```

Link zur Counter-Klasse

Fano-Codierung: Implementierung - Schritt 2

Symbole nach Häufigkeit sortieren

```
# sort the frequencies in descending order
frequencies = dict(sorted(frequencies.items(),
                          key=lambda item: item[1],
                          reverse=True))
```

Fano-Codierung: Implementierung - Schritt 3

Liste in zwei Teile aufteilen

```
def split_frequencies(frequencies):
    total_frequency = sum(frequencies.values())
    cumulative_frequency = 0
    split_index = 0
    index = 0

    # find the index at which the cumulative frequency is greater than
    # or equal to half of the total frequency
    for char, freq in (frequencies.items()):
        cumulative_frequency += freq
        index += 1
        if cumulative_frequency >= total_frequency / 2:
            split_index = index
            break

    # split the frequencies dictionary into two parts
    left_symbols = dict(list(frequencies.items())[:split_index])
    right_symbols = dict(list(frequencies.items())[split_index:])
    return total_frequency,left_symbols,right_symbols
```

Fano-Codierung: Implementierung - Schritt 4

Binäre Zuweisung

```
def generate_codes(node, prefix="", codebook={}):
    if node is not None:
        if node.char is not None:
            codebook[node.char] = prefix
        generate_codes(node.left, prefix + "0", codebook)
        generate_codes(node.right, prefix + "1", codebook)
    return codebook
```

Fano-Codierung: Implementierung - Schritt 5.1

Baum-Klasse anlegen

```
class ShannonFanoNode:
    # Node class for the Shannon-Fano tree with
    # a character, frequency, left and right child
    def __init__(self, char, freq):
        # Initializes the node with a character
        # and its frequency
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        # Defines the less than operator to compare
        # nodes based on their frequency
        return self.freq < other.freq
```

Fano-Codierung: Implementierung - Schritt 5.2

Rekursive Anwendung

```
def build_tree(frequencies):
    # Builds the Shannon-Fano tree from a
    # dictionary of character frequencies
    if len(frequencies) == 1:
        temp_list = list(frequencies.items())
        return ShannonFanoNode(temp_list[0][0], temp_list[0][1])

    total_frequency, left, right = split_frequencies(frequencies)

    node = ShannonFanoNode(None, total_frequency)
    node.left = build_tree(left)
    node.right = build_tree(right)

    return node
```

Optimale Codierung mit Huffman

- Einige Jahre später stellte David A. Huffman (1952) einen anderen Algorithmus vor, der für beliebige Symbolwahrscheinlichkeiten immer einen optimalen Baum erzeugt.
- Während der Shannon-Fano-Baum von Fano durch Unterteilung von der Wurzel zu den Blättern entsteht, arbeitet der Huffman-Algorithmus in umgekehrter Richtung, indem er von den Blättern zur Wurzel vorgeht.

Huffmans Algorithmus

Der Prozess beginnt mit den Blattknoten, die die Wahrscheinlichkeiten des Symbols enthalten, das sie darstellen. Dann werden die beiden Knoten mit der geringsten Wahrscheinlichkeit ausgewählt und ein neuer interner Knoten erstellt, der diese beiden Knoten als Kinder hat. Das Gewicht des neuen Knotens wird auf die Summe der Gewichte der Kinder gesetzt. Anschließend wird der Prozess erneut auf den neuen internen Knoten und die verbleibenden Knoten angewandt (d. h. die beiden Blattknoten werden ausgeschlossen), bis nur noch ein Knoten übrig bleibt, der die Wurzel des Huffman-Baums ist.

Huffman-Codierung: Schritte zur Implementierung

Prioritätswarteschlange der Symbole und Häufigkeiten entwickeln

Erstelle für jedes Symbol einen Blattknoten und füge es in eine Prioritätswarteschlange ein, wobei die Häufigkeit seines Auftretens als Priorität gilt. Das Symbol mit der geringsten Häufigkeit steht also vorne in der Schlange.

Eine Prioritätswarteschlange ist so implementiert, dass ein neu eingefügtes Element auch immer direkt richtig einsortiert wird. Meist wird dies mit einem Heap realisiert.

Zusammenfassen der Knoten mit der geringsten Häufigkeit

Wenn sich mehr als ein Knoten in der Warteschlange befindet:

- Entferne die beiden Knoten mit der geringsten Wahrscheinlichkeit oder Häufigkeit aus der Warteschlange.
- Stelle den Codes, die diesen Knoten bereits zugewiesen sind, 0 bzw. 1 voran.
- Erzeuge einen neuen internen Knoten mit diesen beiden Knoten als Unterknoten und weise die Summe der Wahrscheinlichkeiten der beiden Knoten diesem Knoten zu und füge ihn in die Warteschlange ein.

Rekursion abbrechen

Wenn sich nur noch ein Knoten in der Warteschlange befindet:

- Der verbleibende Knoten ist der Wurzelknoten und der Baum ist vollständig.

Huffman-Codierung: Implementierung

Huffman Baum aufbauen

```
def build_huffman_tree(frequencies):
    # Create a min heap
    heap = [HuffmanNode(char, freq) for char, freq in frequencies.items()]
    heapq.heapify(heap)

    # Go through the tree from the bottom up
    while len(heap) > 1:
        # remove the two smallest nodes
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        # merge them into a new node
        merged_node = HuffmanNode(None, node1.freq + node2.freq)
        merged_node.left = node1
        merged_node.right = node2
        # push the new node back into the heap
        heapq.heappush(heap, merged_node)

    return heap[0]
```

Huffman-Codierung: Implementierung

Huffman Codes generieren

```
def generate_huffman_codes(node, prefix="", codebook={}):
    if node is not None:
        if node.char is not None:
            codebook[node.char] = prefix
        generate_huffman_codes(node.left, prefix + "0", codebook)
        generate_huffman_codes(node.right, prefix + "1", codebook)
    return codebook
```

Huffman-Codierung: Implementierung

Huffman Codes encodieren

```
def encode(message, huffman_dict):
    return ''.join(huffman_dict[char] for char in message)
```

Das Codebook liegt als Dictionary vor und man schlägt einfach jedes Zeichen nach. Dann fügt man die Codes aneinander.

Huffman-Decodierung

Was muss der Empfänger wissen um einen Huffman Code dekodieren zu können?

Huffman-Decodierung

Im Allgemeinen geht es bei der Dekomprimierung einfach darum, den Datenstrom aus Präfixcodes in einzelne Zeichen oder Symbole zu übersetzen, indem der Huffman-Baum Knoten für Knoten durchlaufen wird, während jedes Bit aus dem Eingabestrom gelesen wird (das Erreichen eines Blattknotens beendet zwangsläufig die Suche nach diesem bestimmten Symbol).

Huffman-Decodierung: Implementierung

```
def decode(encoded_message, huffman_dict):
    # create a reverse dictionary for decoding
    reverse_dict = {v: k for k, v in huffman_dict.items()}
    # decode the message
    current_code = ''
    decoded_message = ''
    for bit in encoded_message:
        current_code += bit
        # check if the current code is in the reverse
        # dictionary
        if current_code in reverse_dict:
            decoded_message += reverse_dict[current_code]
            current_code = ''
    return decoded_message
```

Huffman-Decodierung in der Praxis

Bevor dies jedoch geschehen kann, muss der Huffman-Baum irgendwie rekonstruiert werden. Im einfachsten Fall, wenn die Zeichenhäufigkeit ziemlich vorhersehbar ist, kann der Baum im Voraus konstruiert und somit jedes Mal wiederverwendet werden, was allerdings zu Lasten einer gewissen Komprimierungseffizienz geht. Andernfalls müssen die Informationen zur Rekonstruktion des Baums a priori übermittelt werden.

Huffman Baum

Wie kann der Huffman Baum übertragen werden?

Kann man auch ihn codieren?

Naiver Ansatz

Ein naiver Ansatz könnte darin bestehen, dem Datenstrom die Häufigkeiten jedes Zeichens voranzustellen. Daraus kann dann der Huffman Baum erzeugt werden.

Leider kann sich der Overhead in einem solchen Fall auf mehrere Kilobyte belaufen, so dass diese Methode wenig praktischen Nutzen hat.

Codierung des Codebooks

Nehmen wir unser ursprüngliches Huffman-Codebuch:

```
A = 11
B = 0
C = 101
D = 100
```

Es gibt mehrere Möglichkeiten, diesen Huffman-Baum zu kodieren. Wir könnten zum Beispiel jedes Symbol gefolgt von der Anzahl der Bits und dem Code schreiben:

```
('A',2,11), ('B',1,0), ('C',3,101), ('D',3,100)
```

Codierung des Codebooks

Da wir die Symbole in fortlaufender alphabetischer Reihenfolge auflisten, können wir die Symbole selbst weglassen und nur die Anzahl der Bits und den Code angeben:

```
(2,11), (1,0), (3,101), (3,100)
```

Bei einer kanonischen Version haben wir die Gewissheit, dass die Symbole in alphabetischer Reihenfolge angeordnet sind und dass ein späterer Code immer einen höheren Wert hat als ein früherer.

Codierung des Codebooks

Die einzigen Teile, die noch übertragen werden müssen, sind die Bitlängen (Anzahl der Bits) für jedes Symbol. Beachten Sie, dass unser kanonischer Huffman-Baum immer höhere Werte für längere Bitlängen hat und dass alle Symbole mit derselben Bitlänge (C und D) höhere Codewerte für höhere Symbole haben:

```
A = 10  # (Codewert: 2 dezimal, Bits: 2)
B = 0   # (Codewert: 0 dezimal, Bits: 1)
C = 110 # (Codewert: 6 dezimal, Bits: 3)
D = 111 # (Codewert: 7 dezimal, Bits: 3)
```

Codierung des Codebooks

Da zwei der drei Vorgaben bekannt sind, muss nur die Anzahl der Bits für jedes Symbol übertragen werden:

```
2, 1, 3, 3
```

Mit der Kenntnis des kanonischen Huffman-Algorithmus ist es dann möglich, die gesamte Tabelle (Symbol- und Codewerte) nur aus den Bitlängen neu zu erstellen. Nicht verwendete Symbole werden normalerweise mit einer Bitlänge von Null übertragen.

Codierung des Codebooks

Eine andere effiziente Möglichkeit, das Codebuch darzustellen, besteht darin, alle Symbole in aufsteigender Reihenfolge ihrer Bitlängen aufzulisten und die Anzahl der Symbole für jede Bitlänge aufzuzeichnen. Für das oben genannte Beispiel sieht die Kodierung wie folgt aus:

```
(1,1,2), ('B','A','C','D')
```

Das bedeutet, dass das erste Symbol B die Länge 1 hat, dann das Symbol A die Länge 2 und die verbleibenden zwei Symbole (C und D) die Länge 3. Da die Symbole nach Bitlänge sortiert sind, können wir das Codebuch effizient rekonstruieren.

Codierung des Codebooks

Diese Art der Kodierung ist vorteilhaft, wenn nur einige wenige Symbole im Alphabet komprimiert werden sollen. Nehmen wir zum Beispiel an, das Codebuch enthält nur 4 Buchstaben C, O, D und E, jeweils mit der Länge 2. Um den Buchstaben O mit der bisherigen Methode darzustellen, müssen wir entweder viele Nullen hinzufügen:

```
0, 0, 2, 2, 2, 0, ... , 2, ...
```

oder aufzeichnen, welche 4 Buchstaben wir verwendet haben.

Codierung des Codebooks

In jedem Fall wird die Beschreibung länger als die folgende:

```
(0,4), ('C','O','D','E')
```

Das JPEG File Interchange Format verwendet diese Methode der Kodierung, da nur maximal 162 Symbole des 8-Bit-Alphabets, das 256 Zeichen umfasst, im Codebuch enthalten sind.

Quelle: Wikipedia