

Übungsaufgaben II, SBV1

Lukas Fiel, Lisa Panholzer

November 21, 2018

2 Übungsaufgaben II

2.1 Resampling und Interpolation

a) Implementierung Resampling

Die Implementierung des Resampling Filters wurde in diesem Abschnitt anhand der Nearest Neighbour Interpolation umgesetzt. Bevor der Filter ausgeführt wird, wird zuerst der Skalierungsfaktor bei dem Benutzer abgefragt. Gibt der Benutzer einen Wert ein, der über 1.0 liegt, wird das Bild vergrößert. Gibt er einen Wert ein, der unter 1.0 liegt wird das Eingangsbild verkleinert.

In dieser Implementierung wird die Umrechnung der Koordinaten anhand der Variante B umgesetzt. Dies bedeutet, dass der Skalierungsfaktor bereits vor der Neuberechnung der Koordinaten angepasst wird, in dem von diesem 1 subtrahiert wird. Das heißt, die neue Koordinate vom skalierten Bild B wird aus der Multiplikation der Koordinate aus dem Originalbild A um den adaptierten Skalierungsfaktor s' berechnet. Dies hat zur Folge, dass sich die Indizes in der Mitte zentrieren. Der Anfang bzw. das Ende des Bild Arrays bleibt hierbei aber weiterhin unterrepräsentiert.

```
; columns
1
2 import ij.*;
3 import ij.plugin.filter.PlugInFilter;
4 import ij.process.*;
5
6 import java.awt.*;
7
8 import ij.gui.GenericDialog;
9
10
11 public class Resample_ implements PlugInFilter {
12
13     public int setup(String arg, ImagePlus imp) {
14         if (arg.equals("about"))
15             {showAbout(); return DONE;}
16         return DOES_8G+DOES_STACKS+SUPPORTS_MASKING;
17     } //setup
18
19
20
21     public void run(ImageProcessor ip) {
22         byte[] pixels = (byte[]) ip.getPixels();
23         int width = ip.getWidth();
24         int height = ip.getHeight();
25         int tgtRadius = 4;
26
27         int newWidth = width;
28         int newHeight = height;
```

```

30 |     int [][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels , width ,
31 |                           ↪ height);
32 |
33 |     //first request target scale factor from user
34 |     GenericDialog dialog = new GenericDialog("user_input");
35 |     dialog.addNumericField("scale_factor:", 1.0, 2);
36 |     dialog.showDialog();
37 |
38 |     if(dialog.wasCanceled()) {
39 |         return;
40 |     }
41 |
42 |     double tgtScaleFactor = dialog.getNextNumber();
43 |     if(tgtScaleFactor < 0.01 || tgtScaleFactor > 10) {
44 |         return;
45 |     }
46 |
47 |     newWidth = (int)(width * tgtScaleFactor + 0.5);
48 |     newHeight = (int)(height * tgtScaleFactor + 0.5);
49 |
50 |     // variant A of transformation of coordinates
51 |     //double scaleFactorX = newWidth / (double)(width);
52 |     //double scaleFactorY = newHeight / (double)(height);
53 |
54 |     // variant B of transformation of coordinates
55 |     double scaleFactorX = (double)(newWidth - 1.0) / (double)(width - 1.0);
56 |     double scaleFactorY = (double)(newHeight - 1.0) / (double)(height - 1.0)
57 |                           ↪ ;
58 |
59 |     //information output
60 |     System.out.println("tgtScale:" + tgtScaleFactor + ",sX:" + 
61 |                           ↪ scaleFactorX + ",sY:" + scaleFactorY);
62 |     System.out.println("newWidth:" + newWidth + ",newHeight:" + 
63 |                           ↪ newHeight);
64 |
65 |     int [][] scaledImage = new int [newWidth][newHeight];
66 |
67 |     //iterate over all pixel of the scaled image
68 |     for(int x = 0; x < newWidth; x++) {
69 |         for (int y = 0 ; y < newHeight; y++) {
70 |             //calculate new scaled x and y coordinates
71 |             double newX = (double)(x) / scaleFactorX;
72 |             double newY = (double)(y) / scaleFactorY;
73 |
74 |             //calculate new result value
75 |             int resultVal = GetNNinterpolatedValue(inDataArrInt ,
76 |                           ↪ newX, newY, width, height);
77 |
78 |             //set new value
79 |             scaledImage[x][y] = resultVal;
80 |
81 |         }
82 |     }
83 |
84 |     //show new image
85 |     ImageJUtility.showNewImage(scaledImage , newWidth, newHeight , "scaled_img");
86 |
87 | } //run
88 |
89 | void showAbout() {
90 |     IJ.showMessage("About_Template_..." ,
91 |                   "this_is_a_PluginFilter_template\n");
92 | } //showAbout

```

```

89 |
90 |
91     public int GetNNinterpolatedValue(int [][] inImg, double x, double y, int width,
92             → int height) {
93         //round x and y position
94         int xPos = (int) (x + 0.5);
95         int yPos = (int) (y + 0.5);
96
97         //safety check
98         if(xPos >= 0 && xPos < width && yPos >= 0 && yPos < height) {
99             return inImg[xPos][yPos];
100        }
101    }
102
103    public int GetBilinearinterpolatedValue(int [][] inImg, double x, double y, int
104             → width, int height) {
105        //implemented in separate java file
106    }
107
108 } //class Resample_

```

b) Implementierung Bi-Lineare Interpolation

Die Implementierung des Resamplings Filters wurde in diesem Abschnitt anhand der Bilinearen Interpolation umgesetzt. Bevor der Filter ausgeführt wird, wird zuerst der Skalierungsfaktor bei dem Benutzer abgefragt. Gibt der Benutzer einen positiven Wert ein, der über 1.0 liegt, wird das Bild vergrößert. Gibt er einen darunterliegenden Wert ein wird das Eingangsbild verkleinert.

Danach wird die neue Höhe und Breite des Eingangsbildes anhand des Skalierungsfaktor berechnet. Zusätzlich wird der Skalierungsfaktor für die x und y Koordinaten separat berechnet und gespeichert. Anschließend wird anhand einer for-Schleife über alle Pixel des neu angelegten Arrays des skalierten Bildes iteriert.

Neben der neuen Koordinate wird der Pixelwert anhand der Methode GetBilinearInterpolatedValue() berechnet. In dieser werden unterschiedliche Fragmente für die Berechnung des gewichteten Mittelwert aufbereitet. Damit der neue skalare Wert berechnet werden kann, müssen zuerst 4 benachbarten Pixel aus dem Originalbild ermittelt werden. Die Nachbarpixel sind folgende: p1(x+1,y), p2(x,y+1), p3(x+1, y+1) und p4(x,y).

Um den gewichtet Mittelwert für diesen Pixel zu erhalten werden anhand der Kalkulationsfragmente, den benachbarten skalaren Werten (p1-p4) der gewichtete Mittelwert berechnet werden. Anschließend wird dieser Wert an die Methode retourniert und im skalierten Bild an der aktuellen Koordinate

eingefügt.

Test der Implementierung

Um die Implementierung des Resampling Filters anhand der bi-linearen Interpolation zu prüfen, wurden zwei Testbilder inklusive Differenzbilder generiert.

Folgendes Bild wurde um den Faktor 3.0 vergrößert:



Figure 1: Skalierung anhand Faktor 3.0

TODO: Testbild einfügen

Testbild zur bilinearen Interpolation (Skalierungsfaktor 2.0):



Figure 2: Resampling anhand bilinearer Interpolation und Skalierung um Faktor 2.0



Figure 3: Resampling anhand Nearest Neighbor Interpolation und Skalierung um Faktor 2.0



Figure 4: Differenzbild generiert aus Nearest Neighbor und bilinearer Interpolation

Das in Figure 3 dargestellte Bild, stellt die Differenz zwischen dem Testbild das anhand der Nearest Neighbor und der bilinearen Interpolation erstellt wurde. Bei näherer Betrachtung sieht man, dass sich an den Kanten im Bild, weiße Ausprägungen wiederfinden. Dies verdeutlicht, dass die NN Interpolation aufgrund keiner Neuberechnung der Werte, diese nicht so gut wiedergibt. Diese Unterschiede in den skalaren Werten ergeben sich daraus, dass die bilineare Interpolation aufgrund der Verwendung des gewichteten Mittelwerts von 4 Pixeln einen ähnlicheren skalaren Wert ergibt, als bei der NN Interpolation.

```
; columns
1
2 import ij.*;
3 import ij.plugin.filter.PlugInFilter;
4 import ij.process.*;
5
6 import ij.gui.GenericDialog;
```

```

8 | public class ResampleBilineareInterpolation_ implements PlugInFilter {
9 |
10|     public int setup(String arg, ImagePlus imp) {
11|         if (arg.equals("about")) {
12|             showAbout();
13|             return DONE;
14|         }
15|         return DOES_8G + DOES_STACKS + SUPPORTS_MASKING;
16|     } // setup
17|
18|     public void run(ImageProcessor ip) {
19|         byte[] pixels = (byte[]) ip.getPixels();
20|         int width = ip.getWidth();
21|         int height = ip.getHeight();
22|
23|         int[][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels, width,
24|             ↑ height);
25|         int newWidth = width;
26|         int newHeight = height;
27|
28|         // first request target scale factor from user
29|         GenericDialog dialog = new GenericDialog("user_input");
30|         dialog.addNumericField("scale_factor", 1.0, 2);
31|         dialog.showDialog();
32|
33|         // if user has canceled the dialog
34|         if (dialog.wasCanceled()) {
35|             return;
36|         }
37|
38|         //get user input of scale factor
39|         double tgtScaleFactor = dialog.getNextNumber();
40|
41|         // check range
42|         if (tgtScaleFactor < 0.01 || tgtScaleFactor > 10) {
43|             return;
44|         }
45|
46|         //calculate new width and height with scale factor
47|         newWidth = (int) (width * tgtScaleFactor + 0.5);
48|         newHeight = (int) (height * tgtScaleFactor + 0.5);
49|
50|         // calculate scale factor per dimension (variant a)
51|         double scaleFactorX = newWidth / ((double) width);
52|         double scaleFactorY = newHeight / ((double) height);
53|
54|         //information output
55|         System.out.println("tgtScale=" + tgtScaleFactor + "sX=" + scaleFactorX
56|             ↑ + "sY=" + scaleFactorY);
57|         System.out.println("new_width=" + newWidth + "new_height=" +
58|             ↑ newHeight);
59|
60|         int[][] scaledImg = new int[newWidth][newHeight];
61|
62|         // fill new result image --> iterate over result image
63|         for (int x = 0; x < newWidth; x++) {
64|             for (int y = 0; y < newHeight; y++) {
65|                 // calculate new coordinate
66|                 double newX = x / scaleFactorX;
67|                 double newY = y / scaleFactorY;
68|
69|                 //get bilinear interpolated value
70|                 double resultVal = GetBilinearInterpolatedValue(

```

```

69                                     ↪ inDataArrInt , newX, newY, width , height );
70
71         //set new rounded value for current location
72         scaledImg[x][y] = (int) (resultVal + 0.5);
73     }
74 }
75
76 //show new image
77 ImageJUtility.showNewImage(scaledImg , newWidth , newHeight , "scaled-img-(
78                                     ↪ bilinear-interpolation");
79 } // run
80
81 public double GetBilinearInterpolatedValue(int [][] inImg , double x , double y ,
82                                     ↪ int width , int height ) {
83
84     // calculate the delta for x and y
85     double deltaX = x - Math.floor(x);
86     double deltaY = y - Math.floor(y);
87
88     // set calculation fragment
89     int xPlus1 = (int) x + 1;
90     int yPlus1 = (int) y + 1;
91
92     //handling of image edge for x
93     if (x + 1 >= width) {
94         xPlus1 = (int) x;
95     }
96
97     //handling of image edge for y
98     if (y + 1 >= height) {
99         yPlus1 = (int) y;
100    }
101
102    // get 4 neighboring pixels
103    int neighbor1 = inImg [xPlus1][(int) (y)];
104    int neighbor2 = inImg [(int) (x)][yPlus1];
105    int neighbor3 = inImg [xPlus1][yPlus1];
106    int neighbor4 = inImg [(int) (x)][(int) (y)];
107
108    // calculate weighted mean out of neighbors
109    double weightedMean = ((1 - deltaX) * (1 - deltaY) * neighbor4) +
110                                     ↪ deltaX * (1 - deltaY) * neighbor1)
111                                     + ((1 - deltaX) * deltaY * neighbor2) + (deltaX * deltaY
112                                     ↪ * neighbor3);
113
114    return weightedMean;
115 }
116
117 void showAbout() {
118     IJ.showMessage("About_Template_... ", "this_is_a_PluginFilter_template\n"
119                                     ↪ );
120 } // showAbout
121
122 } // class ResampleBilinearInterpolation_

```

c) Implementierung Checker-Board

Die Nearest Neighbor Interpolation bezieht sich in ihrer Berechnung auf keinen neuen skalaren Wert. Es werden jeglich die benachbarten Pixel geprüft,

und der Wert der am Nächsten ist, für den neuen Pixel herangezogen. Die Neuzuweisung schlägt sich auf die Bildqualität wieder, dieser wird bei einer Skalierung pixeliger bzw. kantiger.

Theoretische Überlegung: Da bei dieser Strategie kein komplexer Algorithmus verwendet wird und nur eine Neuzuweisung statt - berechnung stattfindet, ist dieser schneller als die andere Strategie.

Die bilineare Interpolation hingegen bedient sich auch der benachbarten Pixel, berechnet jedoch auf Basis dieser den gewichteten Mittelwert. Dieser Wert ist ähnlicher als zum Beispiel die Zuweisung der NN Interpolation und resultiert daher in einem besseren Bildqualität.

Theoretische Überlegung: Aufgrund der Verwendung eines komplexeren Algorithmus ist die Laufzeit bei diesem auch länger.

2.2 Klassifizierung mittels Kompression

a) Klassifizierung von Texten

Idee

Aus Texten in 8 verschiedenen Sprachen soll mittels Kompression eine Klassifizierung stattfinden. Zur Testzwecken wurden folgende Datensätze als *.txt Dateien in deutsch, englisch, französisch, spanisch, polnisch, ungarisch, bosnisch, und niederländisch vorbereitet:

- Abstract einer wissenschaftlichen Arbeit. Diese hatte den Vorteil dass es eine deutsche und englische Übersetzung gab. Alle weiteren Sprachen wurden aus der englischen Version mittels *google translate* generiert.
- Wörterbuch mit 10000 deutschen Wörtern. Dieser Datensatz wurde mittels *google translate* in alle anderen Sprachen übersetzt.
- Die erste Seite der Datenschutzrichtlinien von Facebook. Da die Datenschutzrichtlinien in sämtlichen Sprachen abrufbar sind, konnte für alle Sprachen ein passender Datensatz gefunden werden.
- Die erste Seite der Datenschutzrichtlinien von Google. Auch hier waren Daten in allen Sprachen verfügbar.
- Ein Witz der aus dem deutschen mittels *google translate* in alle andern Sprachen übersetzt wurde.

Zur Weiterverarbeitung wurden den Sprachen Nummern zugeordnet die man folgender Liste entnehmen kann.

- 1 = bosnisch
- 2 = deutsch
- 3 = englisch
- 4 = spanisch
- 5 = französisch
- 6 = niederländisch
- 7 = polnisch
- 8 = ungarisch

Da nach einer Übersetzung die Texte in verschiedenen Sprachen ungleich viele Buchstaben beinhalten ist auch die Dateigröße unterschiedlich. Dies könnte eventuell rechnerisch berücksichtigt werden. Viel einfacher aber ist es, die letzten Buchstaben jedes langen Textes zu ignorieren und so eine einheitliche Länge des Textes zu gewährleisten. Dies wurde mittels eines shell-Skripts erreicht, welches nur die ersten n Bytes eines Files speichert. So konnte für jeden Text eine Datei erzeugt werden die in allen Sprachen den selben Speicherbedarf hat. Der Verlust der letzten Byte ist bei einer Klassifizierung unwesentlich.

```
: columns
1 #!/bin/bash
2
3
4 # mkdir cutTestData
5 mkdir cutTestData/witzData/
6
7 for filename in TestData/witzData/*.txt; do
8     dd bs=1 count=8200 if="$filename" of="cut$filename"
9
10 done
```

Nach einer solchen Normierung der Texte können diese miteinander verglichen werden. Dazu wurde ein Programm in *Octave* geschrieben (siehe Listing a), welches die Texte der einzelnen Datensätze miteinander vergleicht und in einer Matrix darstellt. Eine qualitativ hochwertige Aussage ob

die Ergebnisse statistische Aussagekraft haben, kann mit 5 Datensätzen nicht getroffen werden. Es ist aber sicherlich ein Trend erkennbar.

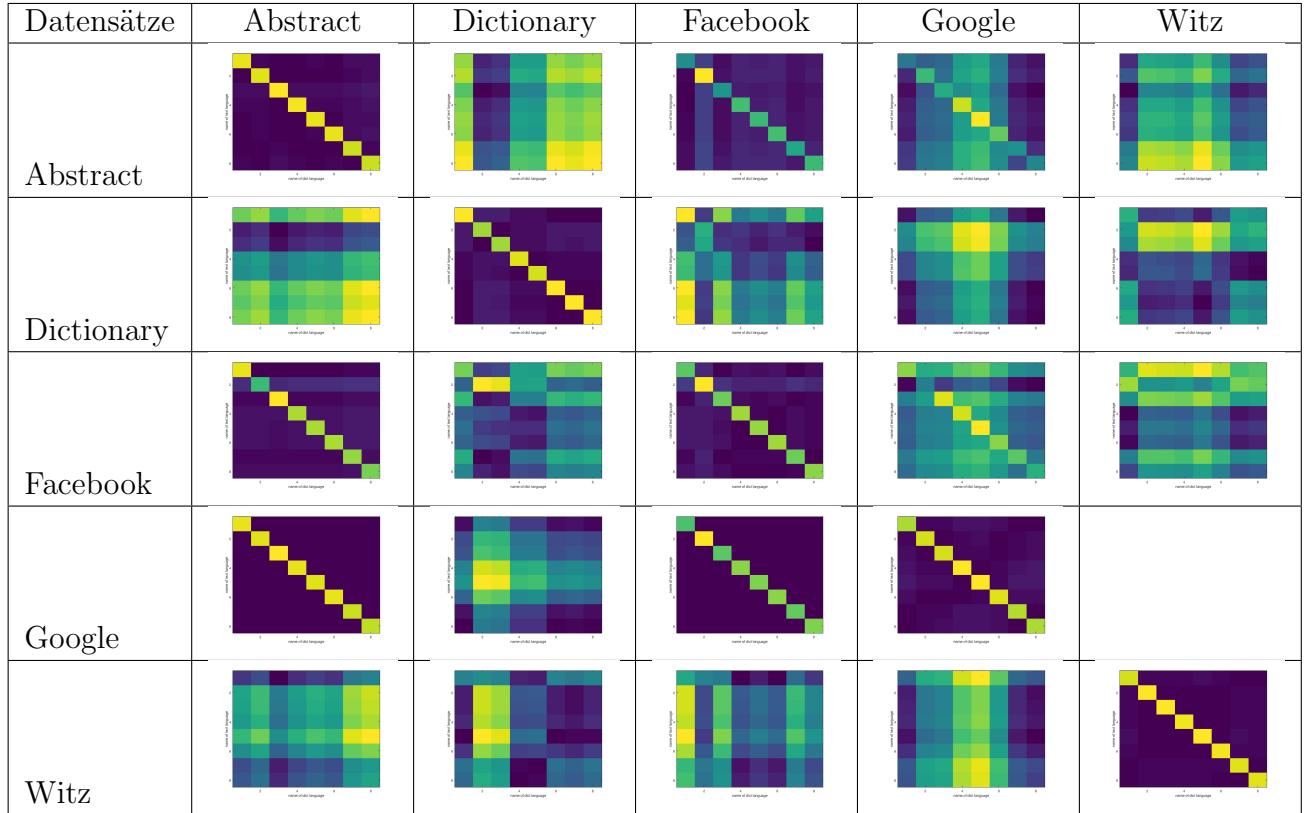


Table 1: Auswertung der Datensätze

Eine Datenauswertung wie in Table 1 lässt deutlich erkennen, dass trivialer Weise eine Symmetrie um die Diagonale besteht und Laufzeit gespart werden könnte wenn diese ausgenutzt würde. In unserer Implementierung wurde darauf aber nicht geachtet.

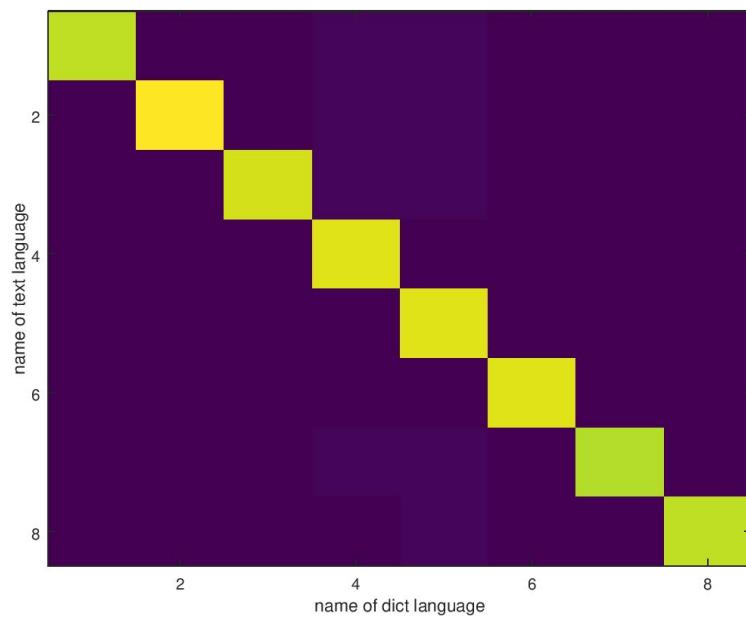


Figure 5: zipDataMatrix

Listing 1: Octave Script zur Darstellung der einzelnen Kompressionsraten.

```

1   ; columns
2
3 # clear console, clear variables, close all open figures
4 clc
5 clear all
6 close all
7
8 pkg load statistics
9
10 # in this example we will use following languages
11 # languages = {"de","en","fr","es","po","un","bo","ne"};
12
13 # constants
14 # please note: copy-pasting windows paths: there are no backslashes in the path
15 folderPath = 'cutTestData/*';
16
17 # initialize result matrix
18 resultMatrix = ones(8);
19
20 # loop through data folders
21 folders = glob(folderPath)
22 for x=1:numel(folders)
23     [~, folderNameI] = fileparts (folders{x})
24     for y=1:numel(folders)
25         [~, folderNameJ] = fileparts (folders{y})
26
27         # just calculate matrix for different data sets
28         # if (!strcmp(folders{x},folders{y}))
29
30             folderPath1 = [folders{x} , '/*'];
31             folderPath2 = [folders{y} , '/*'];
32             filesOffFolder1 = glob(folderPath1);
33             filesOffFolder2 = glob(folderPath2);
34
35             # for every element in data folder
36             for i=1:numel(filesOffFolder1)
37                 [~, nameI] = fileparts (filesOffFolder1{i});
38                 #get file size
39                 [info, err, msg] = stat (filesOffFolder1{i});
40                 file1Size = info.size;
41
42                 #CREATE zip of file
43                 tmpFolderPath = nameI;
44                 mkdir(tmpFolderPath);
45                 copyfile(filesOffFolder1{i},tmpFolderPath);
46                 firstZipName = [nameI, '.zip']
47                 zip(firstZipName,[tmpFolderPath , '/*']);
48                 #get zip size
49                 [info, err, msg] = stat (firstZipName);
50                 file1ZipSize = info.size;
51
52                 # calculate compression rate
53                 file1CompressionRate = file1Size / file1ZipSize;
54
55                 #delete zip and folder as we just need the size for calculation
56                 delete([tmpFolderPath , '/' ,nameI, '.txt']);
57                 rmdir(tmpFolderPath);
58                 delete(firstZipName);
59
60                 for j=1:numel(filesOffFolder2)
61                     [~, nameJ] = fileparts (filesOffFolder2{j});

```

```

62 |     zipName = [nameI, nameJ, '.zip'];
63 |     #create tmp folder in testData folder
64 |     tmpFolderPath = [nameI, nameJ];
65 |     mkdir(tmpFolderPath);
66 |
67 |     #copy filesOfFolder1 to folder
68 |     ['copy_-filesOfFolder1_-' , nameI, '_,-_ nameJ , '_to_-' , tmpFolderPath];
69 |     copyfile(filesOfFolder1{i},tmpFolderPath);
70 |     copyfile(filesOfFolder2{j},tmpFolderPath);
71 |
72 |     # get folder size (add jokeFile size to filesOfFolderlize)
73 |     [info, err, msg] = stat(filesOfFolder2{i});
74 |     file2Size = file1Size + info.size;
75 |
76 |     #zip it and get size of zip
77 |     zip(zipName,[tmpFolderPath,'/*']);
78 |     [info, err, msg] = stat(zipName);
79 |     file2ZipSize = info.size;
80 |
81 |     #calculate compression rate
82 |     compressionRate = file2Size / file2ZipSize ;
83 |
84 |     #calculate expected rate
85 |     expectedfile2ZipSize = file2Size * file1CompressionRate;
86 |
87 |     %Matrix(i,j) = (expectedfile2ZipSize - file2ZipSize) / file2ZipSize;
88 |     kompressionDict = file1Size / file1ZipSize;
89 |     kompressionBoth = file2Size / file2ZipSize;
90 |     kompressionsDelta = abs(kompressionBoth - kompressionDict);
91 |     Matrix(i,j) = kompressionsDelta;
92 |
93 |
94 |     #cleanup - remove tmp folder
95 |     ['remove-' tmpFolderPath];
96 |     delete([tmpFolderPath,'/','nameI','.txt']);
97 |     delete([tmpFolderPath,'/','nameJ','.txt']);
98 |     rmdir(tmpFolderPath);
99 |     delete(zipName);
100 | endfor
101 | endfor
102 |
103 |
104 | resultMatrix = resultMatrix .+ Matrix;
105 | #resultMatrix = (resultMatrix.* Matrix);
106 |
107 | h=figure()
108 | imagesc(Matrix)
109 | view(2)
110 | xlabel("name of dict language");
111 | ylabel("name of text language");
112 | zlabel("compression rates matrix");
113 |
114 | tmpImageFolderName = ["images/",folderNameI];
115 | mkdir(tmpImageFolderName);
116 | cd(tmpImageFolderName);
117 | saveas(h, [folderNameJ,".jpg"], "jpg")
118 | cd("../");
119 |
120 | # else # dont calculate anything if the folders are the same
121 | #     ["skip " folders{x}]
122 | # endif
123 |
124 |

```

```

125 |   endfor
126 | endfor
127 |
128 | g=figure()
129 | imagesc(resultMatrix)
130 | view(2)
131 | xlabel("name of dict language");
132 | ylabel("name of text language");
133 | zlabel("diff of compression rates");
134 | saveas(g, "resultMatrix.jpg","jpg")
135 | 'SUCCESS'

```

b) OPTIONAL – nur für Interessierte/Experten

2.3 Kompression und Code-Transformation

a) Lempel-Ziv Kompression einer Sequenz

Figure a) zeigt die händische Berechnung der Lemper Ziv Kompression. Beim Übertragen ins Protokoll wurde allerdings ein Fehler entdeckt, der in Tabelle 2 korrigiert wurde.

Aufgabe 2.3.a Lempel Ziv Kompression

Zeichenkette: abababbbaaaaabababc<ddd@
 aktuelles Zeichen nächstes Zeichen Ausgabe? Ins Wörterbuch?

a	b	N → a	ab	256
b	a	N → b	ba	257
a	b	y → 258	aba	258
@	b	y → 256 //	abb	259
b	b	N → b	bb	260
b	a	y 257	baa	261
a	a	N → a	aa	262
a	a	y 262	aab	263
b	a	y 257	bab	264
b	a	y 257 y 264	bab	265
c	d	N → c	cc	266
c	d	N → d	cd	267
d	d	y 268	dd	268
d	d	y 268	dda	269

23 Zeichen im Ursprungstext
 14 Zeichen im Resultat

Zeichenkette:
 97, 98, 256, 256, 98, 257, 97, 262, 257, 257, 264, 99, 99,
 ↳ 100, 268

Kompressionsrate: $\frac{23}{14} = 1,64$

aktuelles Zeichen	nächstes Zeichen	Ausgabe (im Wörterbuch?)	ins Wörterbuch!	Speicher
a	b	N → a	ab	256
b	a	N → b	ba	257
a	b	Y → 256	aba	258
a	b	Y → 256	abb	259
b	b	N → b	bb	260
b	a	Y → 257	baa	261
a	a	N → a	aa	262
a	a	Y → 256	aab	263
b	a	Y → 257	bab	264
b	a	Y (257), Y → 264	babc	265
c	c	N → c	cc	266
c	d	N → c	cd	267
d	d	N → d	dd	268
d	d	Y → 268	dda	269
a		N → a		

Table 2: Level Ziv Kompression

In der korrigierten Version ergibt die resultierende Zeichenkette:

97	98	256	256	98	257	97	262	257	264	99	99	100	268	269
----	----	-----	-----	----	-----	----	-----	-----	-----	----	----	-----	-----	-----

$$[H]KompressionsrateC = \frac{23}{15} = 1.5334 \quad (1)$$

b) Huffmann Coding

Die nächste Seite zeigt die händische Berechnung des Huffmann Baums zum gegebenen Beispiel. Die mittlere Codewortlänge liegt dabei bei 1.869bit.

Weiters kann man auf der darauffolgenden Seite die manuelle Berechnung von 6 Testfällen finden. Es kann gesagt werden, dass die Huffmann Kompression sehr gut geeignet wäre um Daten zu komprimieren, die sehr oft gleich sind. Dabei spielt homogenität keine Rolle, sondern rein die Häufigkeit des Auftretens der Sonderfälle. An dieser Stelle sei erwähnt, dass die Information

über solche Sonderfälle nicht verloren geht, sondern einfach mehr Speicher benötigt (verlustfreies Komprimieren).

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
a b a b a b b a a a a b a b a b a b c c d d d a

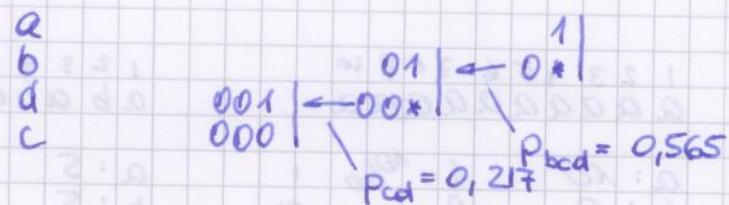
codierung a = 00, b = 01, c = 10, d = 11

23 Zeichen zu 2 bit ergeben eine Länge von 46 bit

Häufigkeiten

$$\left. \begin{array}{l} a: 10 \\ b: 8 \\ c: 2 \\ d: 3 \end{array} \right\} \frac{10}{23} = 0,435 \quad \left. \begin{array}{l} 8/23 = 0,348 \\ 2/23 = 0,087 \\ 3/23 = 0,130 \end{array} \right\} \text{sort} \quad \left\{ \begin{array}{lll} a & | & 1 \\ b & 01 & 2 \\ c & 000 & 4 \\ d & 001 & 3 \end{array} \right.$$

Baum



mittlere Kodewortlänge

$$L = 1 \cdot 10 + 2 \cdot 0,348 + 3 \cdot 0,087 + 3 \cdot 0,130 = 1,782 \text{ bit}$$

$$\text{Kompressionsrate} = \frac{2\text{bit}}{1,782\text{bit}} = 1,122$$

$$\text{Binärlayout: } 10 \cdot 1 + 8 \cdot 2 + 2 \cdot 3 + 3 \cdot 3 = 41 \text{ bit}$$

Zeichen: $\{a, b, c, d\} \rightarrow$ 2bit Darstellung $\{00, 01, 10, 11\}$

1 2 3 4 5 6 7 8 9 10
a b c d a b c d a b

a: 3	$3/10 = 0,3$	1
b: 3	$3/10 = 0,3$	01
c: 2	$2/10 = 0,2$	001
d: 2	$2/10 = 0,2$	000

$$L = 1 \cdot 0,3 + 2 \cdot 0,3 + 3 \cdot 0,2 \cdot 2 \\ = 2,1$$

1 2 3 4 5 6 7 8 9 10
a a a a a a a b c d

a: 7	$7/10 = 0,7$	1
b: 1	$1/10 = 0,1$	01
c: 1	$1/10 = 0,1$	001
d: 1	$1/10 = 0,1$	000

$$L = 1 \cdot 0,7 + 2 \cdot 0,1 + 3 \cdot 0,1 \cdot 2 \\ = 1,5$$

Kompressionsrate: 0,95
→ keine Kompression

Kompressionsrate: 1,5 1,3

1 2 3 4 5 6 7 8 9 10
a a a a a a a a a a a a

a: 10	$1 = 10/10$	1
b: 0	0	01
c: 0	0	001
d: 0	0	000

$$L = 1 \cdot 1 + 2 \cdot 0 + 3 \cdot 0 \cdot 2 \\ = 1$$

Kompressionsrate: 2

1 2 3 4 5 6 7 8 9 10
a b a b a b a b a b

a: 5	$5/10 = 0,5$	1
b: 5	$5/10 = 0,5$	01
c: 0	0	001
d: 0	0	000

$$L = 1 \cdot 0,5 + 2 \cdot 0,5 + 3 \cdot 0 \\ = 1,5$$

Kompressionsrate: 1,3

1 2 3 4 5 6 7 8 9 10
b b b b b c b b b b

a: 0	0	000
b: 9	$9/10 = 0,9$	1
c: 1	$1/10 = 0,1$	01
d: 0	0	001

$$L = 1 \cdot 0,9 + 2 \cdot 0,1 + 3 \cdot 0 \cdot 2 \\ = 1,1$$

Kompressionsrate: 1,82

1 2 3 4 5 6 7 8 9 10
c d c d c d c d c d

a: 0	0	000
b: 0	0	001
c: 5	$5/10 = 0,5$	01
d: 5	$5/10 = 0,5$	1

$$L = 1 \cdot 0,5 + 2 \cdot 0,5 + 3 \cdot 0 = 1,5$$

Kompressionsrate: 1,3

c) Komprimierung einer Sequenz mittels Runlength Coding

Berechnung der Kompressionsrate

Die nachfolgende Sequenz soll anhand von Runlength Coding händisch komprimiert und die Kompressionsrate ausgegeben werden:

01010111100000111100010101111 (30 Stellen, n=2 Symbole:0,1)

Die Sequenz wird von links nach rechts codiert, und anstatt der eigentlichen Zeichen die Häufigkeit dieser ausgegeben. Nach der händischen Komprimierung weist die Frequenz folgende Lauflänge auf:

11111554311114 (14 Stellen)

Daraus ergibt sich folgende Kompressionsrate:

$30/14 = \mathbf{2,14}$

Erweiterung der Symbolmenge

Die Komprimierung von Sequenzen anhand der RLC ist am effektivsten, je homogener der Informationsgehalt ist. Das bedeutet, dass bei vielen unterschiedlichen Zeichen die Komprimierungsmethode nicht mehr sinnvoll angewandt werden kann. Bei sehr kurzen Sequenzen kann es sogar zu einer Erhöhung der Zeichenanzahl in dieser kommen.

Wird nun die Anzahl der Symbole erhöht, muss beachtet werden, dass zusätzlich zu der Häufigkeit des Zeichens noch ein Trennsymbol, eine ID bzw. das Zeichen selbst mitgegeben werden muss. Dies wird anhand eines selbst gewählten Beispiels demonstriert.

Folgende Sequenz wird anhand von RLC komprimiert:

AAZZBBBBBBBCCCCCCCDEEEEEEEFFGHIJJJKLMN

(40 Stellen, n=14 Symbole:A,Z,B,C,D,E,F,G,H,I,J,K,L,M,N)

Die komprimierte Sequenz lautet:

A2Z2B8C7D1E7F2H1I2J3K1L1M1N1 (28 Stellen)

Daraus ergibt sich folgende Kompressionsrate:

$40/28=\mathbf{1,42}$

Wird eine sehr kurze Sequenz mit einer hohen Anzahl an Symbolen komprimiert, kann es aufgrund des hohen Informationsgehalts dazu kommen, dass die codierte Sequenz länger ist, als die Originale.

Folgende Sequenz wird anhand von RLC komprimiert:

AZBBCDEEFFFGHIIJJJKLLLNMOPQRSTU

(30 Stellen, n=21 Symbole: A, Z, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T)

Die komprimierte Sequenz lautet:

A1Z1B2C1D1E2F3G1H1I2J3K1L3M1N1O1P1Q1R1S1T1 (42 Stellen)

Daraus ergibt sich folgende Kompressionsrate:

$30/42 = 0,71$

d) Entropieberechnung

Folgende Sequenz zur Entropieberechnung ist gegeben:

111122661112233334564511211111 (30 Stellen, n=6 Symbole: 1, 2, 3, 4, 5, 6)

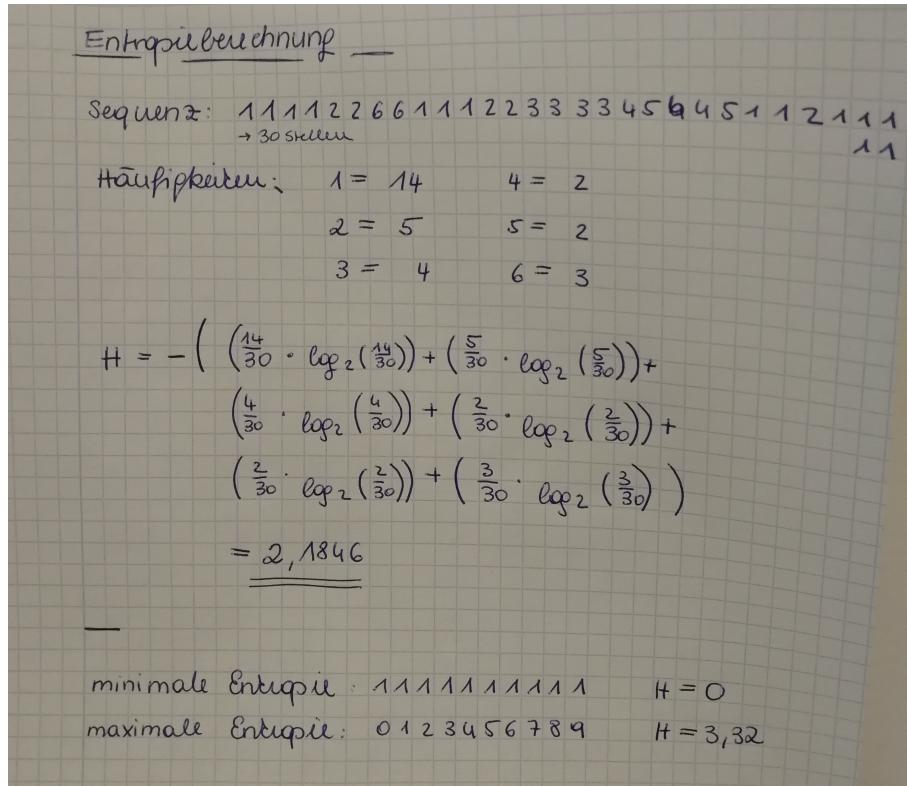


Figure 6: manuelle Entropieberechnung

Nachdem die Häufigkeiten der Symbole erfasst wurde, wurde die Formel der Shannon-Entropie angewandt. Das Ergebnis der Berechnung lautet **2,1846**.

Minimale und Maximale Entropie

Bei folgender 10-stelliger Sequenz ist die Entropie minimal:

Sequenz: 1111111111 (10 Stellen, n=1 Symbol: 1)

$$H = 0$$

Bei folgender 10-stelliger Sequenz ist die Entropie maximal:

Sequenz: 0123456789 (10 Stellen, n=10 Symbole: 0,1,2,3,4,5,6,7,8,9)

$$H = 3,32$$

Auswirkung auf Kompressionsrate

Eine geringe Entropie bewirkt, dass die Sequenz besser komprimierbar ist. Je höher die Entropie, desto schlechter ist die Sequenz komprimierbar. Nicht nur die Auftrittswahrscheinlichkeit hat eine Auswirkung auf die erzielbare Kompressionsrate. Auch die Länge der Sequenz, sowie die Anzahl der Zeichen spielt eine wesentliche Rolle. Je größer diese Faktoren, desto schlechter ist eine Sequenz durch unterschiedliche Kompressionsverfahren bearbeitbar.