

Übungsaufgaben II, SBV1

Lukas Fiel, Lisa Panholzer

November 21, 2018

2 Übungsaufgaben II

2.1 Resampling und Interpolation

Die Implementierung des Resampling Filters und der Checkerboard Darstellung wurde in einem Java-File durchgeführt. Der Code für die Implementierung wird aufgrund der Lesbarkeit nur einmal in Punkt a) im Dokument ausgegeben. Bei einer Ausführung des Filters wird der Skalierungsfaktor und die Größe des Checkerboards in einem Dialog abgefragt. Liegt der Wert des Skalierungsfaktors über 1.0 wird das Bild vergrößert. Liegt der Wert darunter wird dieses verkleinert.

Nach dem der User die Eingaben bestätigt, wird ein skaliertes Bild der Nearest Neighbor- und der Bi-linearen Interpolation ausgegeben. Zusätzlich wird ein Checkerboard ausgegeben, dass beide Interpolationsstrategien zum Vergleich setzt. Die Anzahl der möglichen Segmente pro Reihe wurde auf 20 beschränkt. In den Abschnitten a) - c) werden die jeweils geforderten Abschnitte beschrieben. Die Implementierung findet sich jedoch nur in einem Java-File.

a) Implementierung Resampling

In diesem Abschnitt wird die Implementierung des Resampling Filters mit der Nearest Neighbour Interpolation beschrieben (siehe Code Dokumentation GetNNinterpolatedValue()). Bei dieser Methode wird der nächste Pixelwert berechnet und als neuer skalarer Wert eingefügt.

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 import ij.*;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.*;
4 import ij.gui.GenericDialog;
```

```

6 | public class ResampleInterpolation_ implements PlugInFilter {
7 |
8 |     public int setup(String arg, ImagePlus imp) {
9 |         if (arg.equals("about")) {
10 |             showAbout();
11 |             return DONE;
12 |         }
13 |         return DOES_8G + DOES_STACKS + SUPPORTS_MASKING;
14 |     } // setup
15 |
16 |     public void run(ImageProcessor ip) {
17 |         byte[] pixels = (byte[]) ip.getPixels();
18 |         int width = ip.getWidth();
19 |         int height = ip.getHeight();
20 |
21 |         int newWidth = width;
22 |         int newHeight = height;
23 |
24 |         int[][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels, width,
25 |             ↪ height);
26 |
27 |         // first request target scale factor from user
28 |         GenericDialog dialog = new GenericDialog("user_input");
29 |         dialog.addNumericField("scale_factor:", 1.0, 2);
30 |         //second request checker board size from user
31 |         dialog.addNumericField("checker_board_size:", 4.0, 2);
32 |         dialog.showDialog();
33 |
34 |         if (dialog.wasCanceled()) {
35 |             return;
36 |         }
37 |
38 |         double tgtScaleFactor = dialog.getNextNumber();
39 |         if (tgtScaleFactor < 0.01 || tgtScaleFactor > 10) {
40 |             return;
41 |         }
42 |
43 |         int segments = (int) (dialog.getNextNumber());
44 |         if (segments < 0.01 || tgtScaleFactor > 20) {
45 |             return;
46 |         }
47 |
48 |         newWidth = (int) (width * tgtScaleFactor + 0.5);
49 |         newHeight = (int) (height * tgtScaleFactor + 0.5);
50 |
51 |         // variant A of transformation of coordinates
52 |         // double scaleFactorX = newWidth / (double)(width);
53 |         // double scaleFactorY = newHeight / (double)(height);
54 |
55 |         // variant B of transformation of coordinates
56 |         double scaleFactorX = (double) (newWidth - 1.0) / (double) (width - 1.0)
57 |             ↪ ;
58 |         double scaleFactorY = (double) (newHeight - 1.0) / (double) (height -
59 |             ↪ 1.0);
60 |
61 |         // information output
62 |         System.out.println("tgtScale:" + tgtScaleFactor + ", sX:" + scale
63 |             ↪ factorX + ", sY:" + scaleFactorY);
64 |         System.out.println("new_width:" + newWidth + ", new_height:" + new
65 |             ↪ Height);
66 |
67 |         // arrays for the two different interpolated images
68 |         int[][] scaledImageNN = new int[newWidth][newHeight];
69 |         int[][] scaledImageBI = new int[newWidth][newHeight];

```

```

65 |
66 |     // variables for checkerboard
67 |     int segmentsWidth = newWidth / segments;
68 |     int segmentsHeight = newHeight / segments;
69 |     int[][] scaledCheckerBoard = new int[newWidth][newHeight];
70 |
71 |     // iterate over all pixel of the scaled images
72 |     //variable for x-axis checker board switch
73 |     boolean switchX = false;
74 |     for (int x = 0; x < newWidth; x++) {
75 |         if (x % segmentsWidth == 0) {
76 |             if (switchX == true) {
77 |                 switchX = false;
78 |             } else {
79 |                 switchX = true;
80 |             }
81 |         }
82 |
83 |         //variable for y-axis checker board switch
84 |         boolean switchY = false;
85 |         for (int y = 0; y < newHeight; y++) {
86 |             if (y % segmentsHeight == 0) {
87 |                 if (switchY == true) {
88 |                     switchY = false;
89 |                 } else {
90 |                     switchY = true;
91 |                 }
92 |             }
93 |
94 |             // calculate new scaled x and y coordinates
95 |             double newX = (double) (x) / scaleFactorX;
96 |             double newY = (double) (y) / scaleFactorY;
97 |
98 |             // calculate new result values
99 |             int resultValNN = GetNNinterpolatedValue(inDataArrInt ,
100 |                                         ↪ newX, newY, width, height);
101 |             double resultValBI = GetBilinearinterpolatedValue(
102 |                                         ↪ inDataArrInt, newX, newY, width, height);
103 |
104 |             // set new values for NN and BI
105 |             scaledImageNN[x][y] = resultValNN;
106 |             scaledImageBI[x][y] = (int) (resultValBI + 0.5);
107 |
108 |             //set new values for checker board
109 |             if (switchY == switchX) {
110 |                 scaledCheckerBoard[x][y] = resultValNN;
111 |             } else {
112 |                 scaledCheckerBoard[x][y] = (int) (resultValBI +
113 |                                         ↪ 0.5);
114 |             }
115 |
116 |             // show new image
117 |             ImageJUtility.showNewImage(scaledImageNN, newWidth, newHeight, "scaled-
118 |                                         ↪ img:nearest_neighbour_interpolation");
119 |             ImageJUtility.showNewImage(scaledImageBI, newWidth, newHeight, "scaled-
120 |                                         ↪ img:bilinear_interpolation");
121 |             ImageJUtility.showNewImage(scaledCheckerBoard, newWidth, newHeight, "
122 |                                         ↪ scaled_img:checkerboard_of_NN_and_BI");

```

```

123         IJ.showMessage("About_Template_...", "this_is_a_PluginFilter_template\n"
124             ↵ );
125     } // showAbout
126
127     // calculates nearest neighbor interpolation
128     public int GetNNinterpolatedValue(int [][] inImg, double x, double y, int width,
129             ↵ int height) {
130         // round x and y position
131         int xPos = (int) (x + 0.5);
132         int yPos = (int) (y + 0.5);
133
134         // safety check
135         if (xPos >= 0 && xPos < width && yPos >= 0 && yPos < height) {
136             return inImg[xPos][yPos];
137         }
138         return 0;
139     }
140
141     // calculates bilinear interpolation
142     public double GetBilinearinterpolatedValue(int [][] inImg, double x, double y,
143             ↵ int width, int height) {
144         // calculate the delta for x and y
145         double deltaX = x - Math.floor(x);
146         double deltaY = y - Math.floor(y);
147
148         // set calculation fragment
149         int xPlus1 = (int) x + 1;
150         int yPlus1 = (int) y + 1;
151
152         // handling of image edge for x
153         if (x + 1 >= width) {
154             xPlus1 = (int) x;
155         }
156
157         // handling of image edge for y
158         if (y + 1 >= height) {
159             yPlus1 = (int) y;
160         }
161
162         // get 4 neighboring pixels
163         int neighbor1 = inImg[xPlus1][(int) (y)];
164         int neighbor2 = inImg[(int) (x)][yPlus1];
165         int neighbor3 = inImg[xPlus1][yPlus1];
166         int neighbor4 = inImg[(int) (x)][(int) (y)];
167
168         // calculate weighted mean out of neighbors
169         double weightedMean = ((1 - deltaX) * (1 - deltaY) * neighbor4) +
170             ↵ deltaX * (1 - deltaY) * neighbor1
171             + ((1 - deltaX) * deltaY * neighbor2) + (deltaX * deltaY
172             ↵ * neighbor3);
173
174         return weightedMean;
175     }
176 }
177 // class ResampleInterpolation_

```

b) Implementierung biineare Interpolation

In diesem Abschnitt wird die Implementierung des Resampling Filters mit der bilinearen Interpolation beschrieben (siehe Code Dokumentation GetBi-

`linearinterpolatedValue()).`

Nach der Eingabe des User Inputs 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 und den benachbarten skalaren Werten ($p1-p4$) der gewichtete Mittelwert berechnet . Daraufhin 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 (beider Interpolationssstrategien) zu prüfen, wurden zwei Testbilder inklusive Differenzbilder generiert.

Testbilder (Skalierungsfaktor 2.0):



Figure 1: Resampling anhand bilinearer Interpolation und Skalierung um Faktor 2.0



Figure 2: Resampling anhand Nearest Neighbor Interpolation und Skalierung um Faktor 2.0



Figure 3: Differenzbild generiert aus Nearest Neighbor und bilinearer Interpolation

Figure 3 visualisiert die Differenz zwischen den Testbildern der Nearest Neighbor und der bilinearen Interpolation. Bei näherer Betrachtung sieht man, dass sich an den Umrissen von Lena im Bild weiße Ausprägungen wiederfinden. Diese weißen skalaren Werte bedeutet, dass es hier zwischen den zwei Bildern Unterschiede gibt. 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.

c) Implementierung Checker-Board

Das Checkerboard wurde innerhalb des Resample Filters implementiert und bedient sich der neuberechneten Werte. Die skalaren Werte, die aus den

beiden Methoden berechnet werden, werden in einem neuen Array für das skalierte Checkerboard abwechselnd eingetragen.

Charakterisierung der Interpolationsstrategien

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. Dieser Effekt tritt dann sehr deutlich auf, wenn das Bild zuerst stark verkleinert und anschließend wieder vergrößert wird (0.5 ; 10).

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 einer 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 bosnisch, deutsch, englisch, spanisch, französisch, niederländisch, polnisch und ungarisch aufbereitet:

- Abstract einer wissenschaftlichen Arbeit. Diese hat den Vorteil, dass es eine deutsche und englische Übersetzung gibt. Alle weiteren Sprachen wurden aus der englischen Version mittels *google translate* generiert.
- Wörterbuch mit 10000 deutschen Wörtern. Dieser Datensatz wurde mittels *google translate* in die anfangs erwähnten 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 anderen Sprachen übersetzt wurde.

Zur Weiterverarbeitung wurden den Sprachen Nummern zugeordnet. Folgende Liste gibt die Zuordnung wieder:

- 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 nicht dieselbe Anzahl an Buchstaben beinhalten, variiert auch die Dateigröße. Dies könnte eventuell rechnerisch berücksichtigt werden. Es wurde jedoch die Variante gewählt, 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 done
10
```

Nach einer solchen Normierung der Texte können diese nun 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.

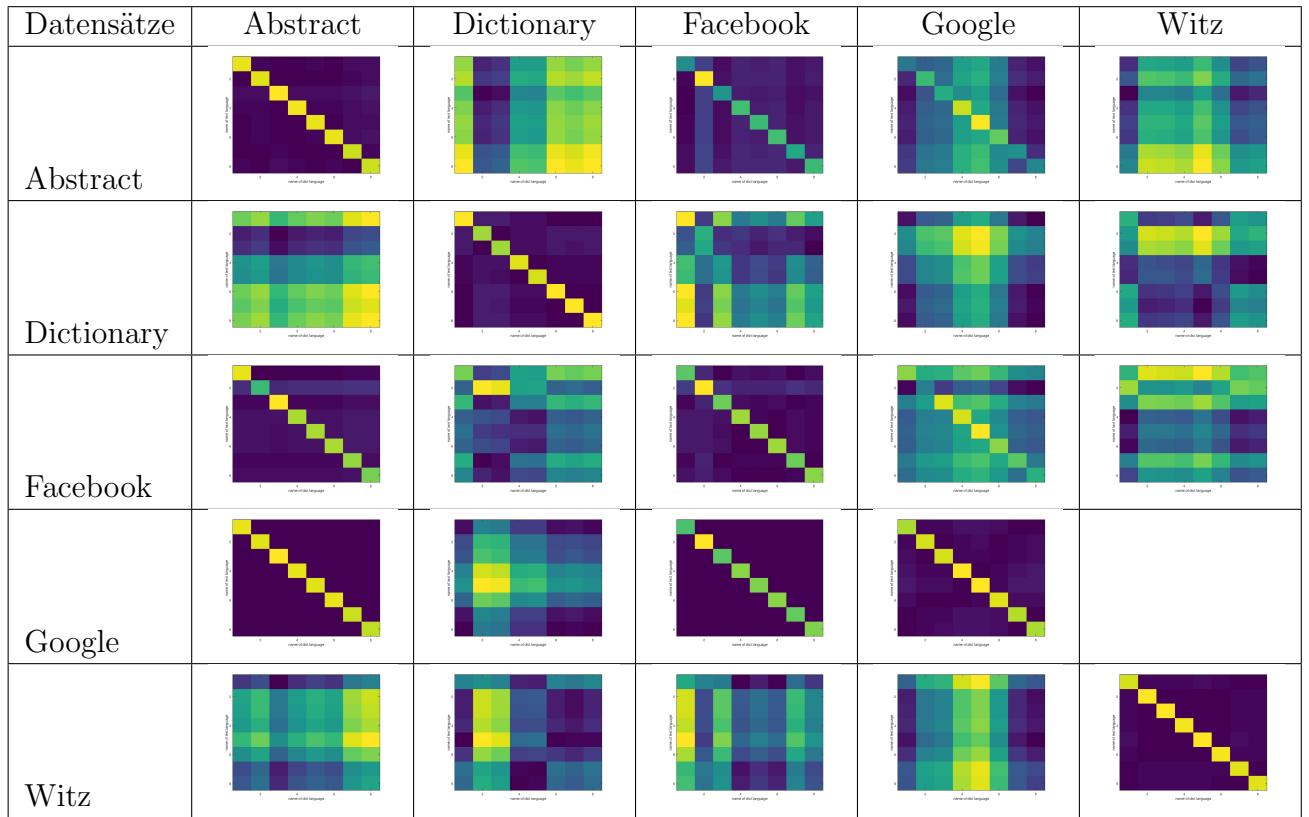


Table 1: Auswertung der Datensätze

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.

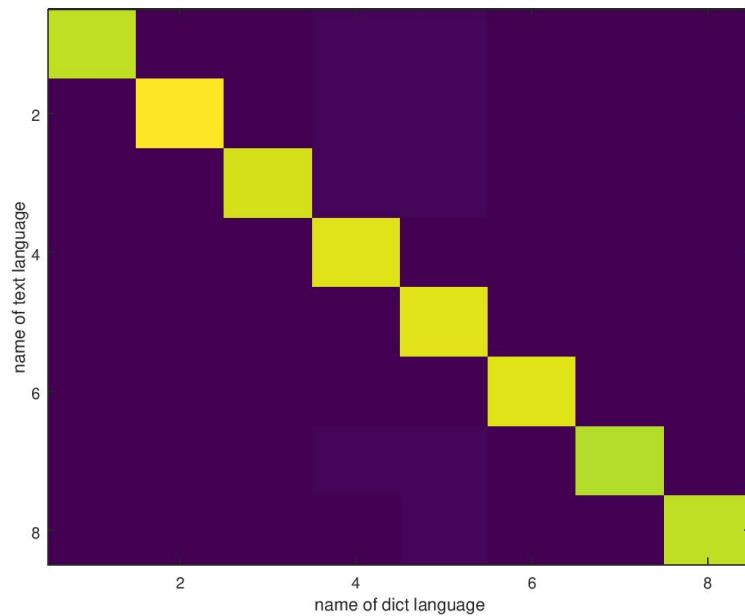


Figure 4: zipDataMatrix

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             filesOfFolder1 = glob(folderPath1);
33             filesOfFolder2 = glob(folderPath2);
34
35             # for every element in data folder
36             for i=1:numel(filesOfFolder1)
37                 [~, nameI] = fileparts (filesOfFolder1{i});
38                 #get file size
39                 [info, err, msg] = stat (filesOfFolder1{i});
40                 file1Size = info.size;
41
42                 #CREATE zip of file
43                 tmpFolderPath = nameI;
44                 mkdir(tmpFolderPath);
45                 copyfile(filesOfFolder1{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(filesOfFolder2)
61                     [~, nameJ] = fileparts (filesOfFolder2{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'

```

2.3 Kompression und Code-Transformation

a) Lempel-Ziv-Welch Kompression einer Sequenz

Figure a) zeigt die händische Berechnung der Lempel-Ziv-Welch 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 wird (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

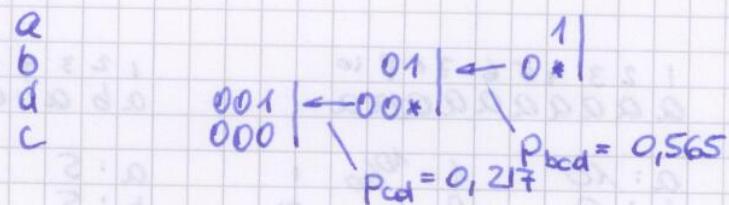
codierung a = 00, b = 01, c = 10, d = 11

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

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

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

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

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)

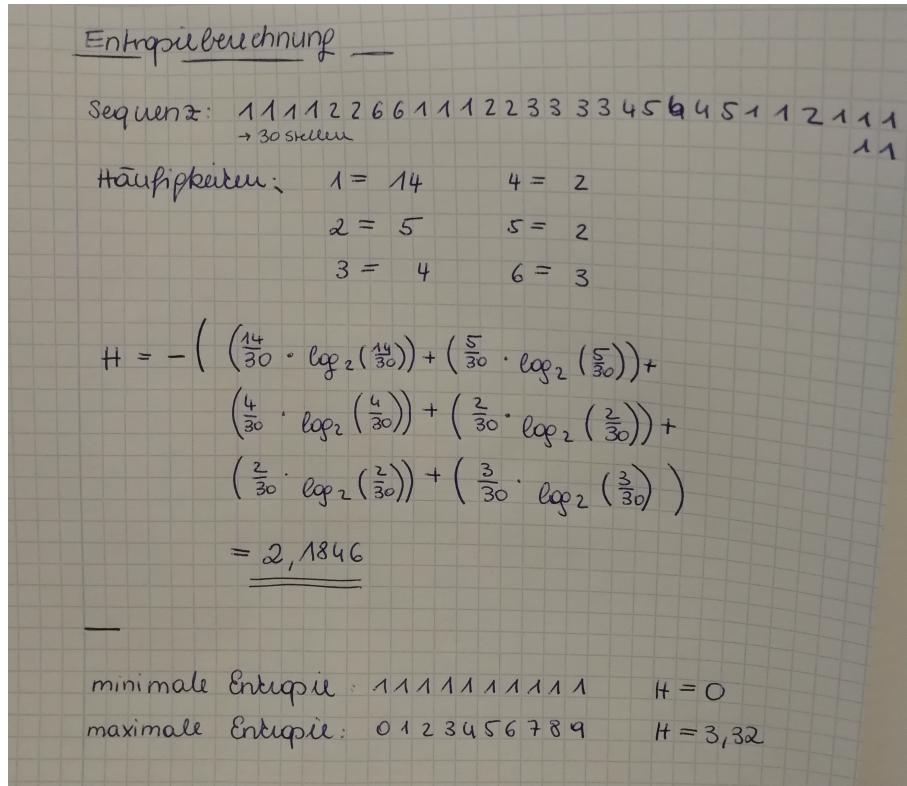


Figure 5: manuelle Entropieberechnung

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.