

SBV1, Signal- und Bildverarbeitung – WS 2022

Übungsabgabe 2

Rudolf Hofmeister / David Lang

6. Dezember 2022

Zusammenfassung

Resampling und Interpolation, Klassifizierung mittels Kompression, Kompression und Code-Transformation.

2.1 Resampling und Interpolation

a) Lösungsidee und Strategien

Beim Resampling von Bildern werden die Bildgrößen um einen Wert verändert. Eine einfache Variante ist, wenn dieser Wert ein ganzzahliges Vielfaches von Höhe und Breite ist, dann werden die Originalpositionen um dieses Vielfache verschoben, dabei entstehen bei einer Vergrößerung Lücken zwischen den Originalwerten. Beim Resampling sind zwei Dinge zu beachten, Die Strategie zur Koordinaten Transformation und das Berechnen der neuen Werte.

Strategien zur Koordinaten-Transformation

Strategie A

Das direkt skalierte Mapping der alten Indizes auf die des neuen Arrays führt zu Über- oder Unterrepräsentationen bei den Pixel. Bei der Strategie A: $b = a - s$, wobei b die Position im neuen Array, a die Position im alten Array und s der Skalierungsfaktor ist, kommt es bei 1-indizierten Array zur Überrepräsentation am Anfang und zur Unterrepräsentation am Ende. Bei 0-indizierten Arrays verhält es sich genau umgekehrt.

Strategie B

Um die verschobene Repräsentation von Strategie A zu korrigieren, wird sich bei der Umverteilung am Index des ersten und des letzten Pixel orientiert. Soll heißen, die Position am ersten und letzten Pixel treffen aufeinander. Es bleibt ein Nachteil, dass nun sowohl am Anfang wie auch am Ende eine Unterrepräsentation auftritt.

$$s' = \frac{w_b - 1}{w_a - 1} \tag{2.1}$$

$$b = a * s' \quad (2.2)$$

Beispiel für Strategie B (siehe Tab. 2.1):

Sei $w_a = 10$ und $w_b = 30$:

$$s' = \frac{30 - 1}{10 - 1} \quad (2.3)$$

$$s' = 3.\bar{2}$$

$$b = a * 3.\bar{2}$$

Strategie C

Um die Nachteile aus Strategie B zu umgehen, werden die Indizes mit einem Offset versehen. Das hat zur Folge, dass nun nicht mehr der erste und letzte Index direkt aufeinandertreffen, sondern die äußeren Ränder der Arrays überlappen jetzt und damit wurde eine bestmögliche Deckung erreicht.

$$r_a = \frac{1}{2 \cdot w_a} + \frac{a}{w_a} \quad (2.4)$$

$$b = \frac{2 \cdot r \cdot w_b - 1}{2} \quad (2.5)$$

Beispiel für Strategie C (siehe Tab. 2.1):

Sei $w_a = 10$ und $w_b = 30$:

$$r_a = \frac{1}{2 \cdot 10} + \frac{a}{10} \quad (2.6)$$

$$b = \frac{2 \cdot r_a \cdot 30 - 1}{2}$$

Tabelle 2.1: Indizes der Strategien A, B und C betrachtet an einem eindimensionalem Array, Skalierungsfaktor 3

Original	Strategie A	Strategie B	Strategie C
0	0	0	1
1	3	3, $\bar{2}$	4
2	6	6, $\bar{4}$	7
3	9	9, $\bar{6}$	10
4	12	12, $\bar{8}$	13
5	15	16, $\bar{1}$	16
6	18	19, $\bar{3}$	19
7	21	22, $\bar{5}$	22
8	24	25, $\bar{7}$	25
9	27	29	28

Für die Implementierung wird Strategie B angewandt, weil Strategie C noch nicht funktioniert.

Nearest Neighbour

Sei m das Offset von x und n das Offset von y , dann gilt es einen Wert vom Pixel an der Stelle $P(x_m, y_n)$ zu finden, wobei $x_m = x + m$ und $y_n = y + n$ ist. Da x_m zwischen x und $x + 1$, und y_n zwischen y und $y + 1$ liegt kann es entweder genau in der Mitte der beiden liegen oder näher bei einem der Nachbarn. Bei der Linearen Interpolation wird schlicht durch ganzzahliges Runden von x_m und y_n der nähere Nachbar identifiziert und dessen Farbwert als neuer Farbwert abgespeichert. Die Implementierung dazu erfolgte in der Vorlesung.

Source Code

Bei der Benutzereingabe wird ein Dropdown ausgegeben, welches Verfahren zum Resampling verwendet werden soll. Ein Skalierungsfaktor > 10 wird auf 10 begrenzt.

```

1 public void run(ImageProcessor ip) {
2     byte[] pixels = (byte[])ip.getPixels();
3     int width = ip.getWidth();
4     int height = ip.getHeight();
5     int[][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels, width, height);
6     double[][] inDataArrDbl = ImageJUtility.convert.ToDoubleArr2D(inDataArrInt, width,
7         height);
8     double scaleFactor = 2.0;
9     // let the user enter factor
10    GenericDialog gd = new GenericDialog("median filter config");
11    gd.addNumericField("scale factor (max 10)", scaleFactor, 4);
12    String[] choices = {"Nearest Neighbour", "Bilinear", "Checkerboard", "Difference"
13    };

```

```

13 gd.addChoice("Method", choices, choices[3]);
14 String choice;
15 gd.showDialog();
16 if (!gd.wasCanceled()) {
17     scaleFactor = gd.getNextNumber();
18     choice = gd.getNextChoice();
19 } else {
20     return;
21 }
22 if (scaleFactor > 10.0) scaleFactor = 10.0;
23
24 int tgtWidth = (int) Math.round(width * scaleFactor);
25 int tgtHeight = (int) Math.round(height * scaleFactor);
26
27 double[][] resampledImage = getResampledImage(inDataArrDbl, width, height,
28     tgtWidth, tgtHeight, false);
28 double[][] resampledBilinearImage = getResampledImage(inDataArrDbl, width, height,
29     tgtWidth, tgtHeight, true);
30
31 if (choice == "Nearest Neighbour") {
32     ImageJUtility.showNewImage(resampledImage, tgtWidth, tgtHeight, "resized with
33         factor " + scaleFactor);
34 } else if (choice == "Bilinear") {
35     ImageJUtility.showNewImage(resampledBilinearImage, tgtWidth, tgtHeight, "resized
36         bilinear with factor " + scaleFactor);
37 } else if (choice == "Checkerboard") {
38     ResizedImage checkerBoard = ImageTransformationFilter.getCheckerBoard(
39         ImageJUtility.convertToIntArr2D(resampledImage, tgtWidth, tgtHeight),
40         ImageJUtility.convertToIntArr2D(resampledBilinearImage, tgtWidth, tgtHeight)
41         ,
42         tgtWidth, tgtHeight, 2);
43     ImageJUtility.showNewImage(checkerBoard.getImage(), tgtWidth, tgtHeight, "
44         Checkerboard with 2 resized images with " + scaleFactor);
45 } else if (choice == "Difference") {
46     double[][] differenceImage = getDifferenceOfImages(resampledImage,
47     resampledBilinearImage, tgtWidth, tgtHeight);
48     ImageJUtility.showNewImage(differenceImage, tgtWidth, tgtHeight, "difference
49         between NN and Bilinear ");
50 }
51 } //run

```

Funktion zum Berechnen des Nearest Neighbour.

```

1 public double getNNInterpolatedValue(double[][] inImg, int width, int height, double
2     posX, double posY) {
3     // e.g. get value of inImg at position (posX, posY) (2.2, 3.7) => inImg[2][4]
4     int posXint = (int) (posX + 0.5); // method to round a double value without Math library
5     int posYint = (int) (posY + 0.5);
6
7     if (posXint < 0) posXint = 0;
8     if (posYint < 0) posYint = 0;
9     if (posXint >= width) posXint = width - 1;
10    if (posYint >= height) posYint = height - 1;
11
12    return inImg[posXint][posYint];
13 }

```

Es wurden die Strategien A, B und C implementiert, aber C funktioniert noch nicht

ordnungsgemäß, daher wurde Strategie B gewählt.

```

1 private double[][] getResampledImage(double[][] inImg, int width, int height, int
2   tgtWidth, int tgtHeight, boolean b) {
3
4   // this is strategy A
5   //   double scaleFactorW = tgtWidth / (double) width;
6   //   double scaleFactorH = tgtWidth / (double) height;
7   //   for (int x = 0; x < width; x++) {
8   //     for (int y = 0; y < height; y++) {
9   //       double posX = x * scaleFactorW;
10  //      double posY = y * scaleFactorH;
11  //      // e.g. I(2,3) scalefactor 4.0 I'(8,12)
12  //      resultImage [(int) Math.round(posX)][(int) Math.round(posY)] =
13  //        getInterpolatedValue(inImg, width, height, x, y);
14  //     }
15  //   }
16  // this is strategy B
17  double scaleFactorW = (tgtWidth - 1.0) / (width - 1.0);
18  double scaleFactorH = (tgtHeight - 1.0) / (height - 1.0);
19  System.out.println("scale X = " + scaleFactorW + " scale Y = " + scaleFactorH);
20  for (int x = 0; x < tgtWidth; x++) {
21    for (int y = 0; y < tgtHeight; y++) {
22      double posX = x / scaleFactorW;
23      double posY = y / scaleFactorH;
24      if (b) {
25        resultImage[x][y] = getBilinearInterpolatedValue(inImg, width, height, posX,
26          posY);
27      } else {
28        resultImage[x][y] = getNNInterpolatedValue(inImg, width, height, posX, posY)
29      }
30    }
31  }
32  // this is strategy C – doesn't work yet
33  //   double scaleFactorW = tgtWidth / (double) width;
34  //   double scaleFactorH = tgtWidth / (double) height;
35  //   for (int x = 0; x < tgtWidth; x++) {
36  //     for (int y = 0; y < tgtHeight; y++) {
37  //       int aX = (int)(x / scaleFactorW);
38  //       int aY = (int)(y / scaleFactorH);
39  //       double offsetX = aX / (double) width;
40  //       double offsetY = aY / (double) height;
41  //       double rX = (1.0 / (2.0 * width)) + offsetX;
42  //       double rY = (1.0 / (2.0 * height)) + offsetY;
43  //       double posX = (2.0 * rX * (double) tgtWidth - 1.0) / 2.0;
44  //       double posY = (2.0 * rY * (double) tgtHeight - 1.0) / 2.0;
45  //       if (b) {
46  //         resultImage[x][y] = getBilinearInterpolatedValue (inImg, width, height,
47  //           posX, posY);
48  //       } else {
49  //         resultImage[x][y] = getNNInterpolatedValue(inImg, width, height, posX,
50  //           posY);
51  //       }
52  //     }
53  //   }
54  // }
55  // }
56  // }
57  // }
58  // }
59  // }
60  // }
61  // }
62  // }
63  // }
64  // }
65  // }
66  // }
67  // }
68  // }
69  // }
70  // }
71  // }
72  // }
73  // }
74  // }
75  // }
76  // }
77  // }
78  // }
79  // }
80  // }
81  // }
82  // }
83  // }
84  // }
85  // }
86  // }
87  // }
88  // }
89  // }
90  // }
91  // }
92  // }
93  // }
94  // }
95  // }
96  // }
97  // }
98  // }
99  // }
100 // }
```

```

50 //      }
51 //    }
52
53 return resultImage;
54 }
```

b) Bilineare Interpolation - Lösungsidee

Bei der Bilinearen Interpolation wird nicht stur zwischen links-rechts und oben-unten entschieden, sondern es werden die Werte aller vier Nachbarspositionen als gewichtetes Mittel in die Berechnung des neuen Farbwert mit einbezogen, wobei als Gewicht das Offset dient. Das ergibt eine viel feinere Farbverteilung. Ein Pixel das beispielsweise zwischen Schwarz und Weiß zu liegen kommt, wird nicht Schwarz oder Weiß wie bei der Nearest Neighbour Interpolation, sondern Grau. Die Berechnung aus Folie 9 wird dazu implementiert werden. Die Multiplikationen welche einen δ -Wert enthalten, sind die Gewichtungen (siehe Gleichung 2.7). Für die Bilineare Interpolation müssen 3 Berechnungen gemacht werden, zwei jeweils auf den Punkten $P(x, y)$ und $P(x + 1, y)$, sowie zwischen $P(x, +1)$ und $P(x + 1, y + 1)$ die zwei parallele Gerade beschreiben. Mit der 3. Berechnung nimmt man nun die interpolierten Ergebnisse der beiden Geraden und interpoliert zwischen diesen Ergebnissen noch einmal und erhält den Ergebniswert für das neue Pixel.

$$\begin{aligned}
 g(x + \delta_x, y + \delta_y) = & (1 - \delta_x)(1 - \delta_y)g(x, y) \\
 & + \delta_x(1 - \delta_y)g(x + 1, y) \\
 & + (1 - \delta_x)\delta_yg(x, y + 1) \\
 & + \delta_x\delta_yg(x + 1, y + 1)
 \end{aligned} \tag{2.7}$$

```

1 public double getBilinearInterpolatedValue(double[][] inImg, int width, int height,
2                                           double posX, double posY) {
3     //Exercise 2.1b
4     int posXint = (int) (posX);
5     int posYint = (int) (posY);
6     // out of Range handling
7     if (posXint < 0) posXint = 0;
8     if (posYint < 0) posYint = 0;
9     if (posXint >= width - 1) posXint = width - 2;
10    if (posYint >= height - 1) posYint = height - 2;
11
12    double deltaX = posX - posXint;
13    double deltaY = posY - posYint;
14
15    double result = (1 - deltaX) * (1 - deltaY) * inImg[posXint][posYint]
16        + deltaX * (1 - deltaY) * inImg[posXint + 1][posYint]
17        + (1 - deltaX) * deltaY * inImg[posXint][posYint + 1]
18        + deltaX * deltaY * inImg[posXint + 1][posYint + 1];
19
20    return result;
21 }
```

```

1 private double[][] getDifferenceOfImages(double[][] firstImage, double[][] secondImage, int width, int height) {
2     double[][] resultImage = new double[width][height];
3     for (int i = 0; i < width; i++) {
4         for (int j = 0; j < height; j++) {
5             resultImage[i][j] = firstImage[i][j] - secondImage[i][j];
6         }
7     }
8     return resultImage;
9 }
```

Ergänzend - Kubische Interpolation

Bei der Bilinearen Interpolation betrachtet man die direkten Nachbarn, dieses Verfahren kann man erweitern, wenn man etwas weiter blickt, nämlich auf die direkten Nachbarn und deren direkte Nachbarn (4x4 Raster). Da der Verlauf dieser Werte nicht mehr linear sein muss, ergibt sich nun statt einer Geraden eine Kurve (Spline) im dreidimensionalem Raum. Das Ergebnis-Spline (gelb) braucht wiederum 4 Splines in den Ebenen aus denen es berechnet werden kann (siehe Abb. 2.6). Dafür müssen 4 polynomiale Gleichungen berechnet werden. Als Fazit bekommt man zwar ein noch feingranulares Resampling, aber zu einem sehr hohen Preis (Rechenleistung).

$$Spline := \sum_{i=1}^n = a_i \cdot x^{i-1} \quad (2.8)$$

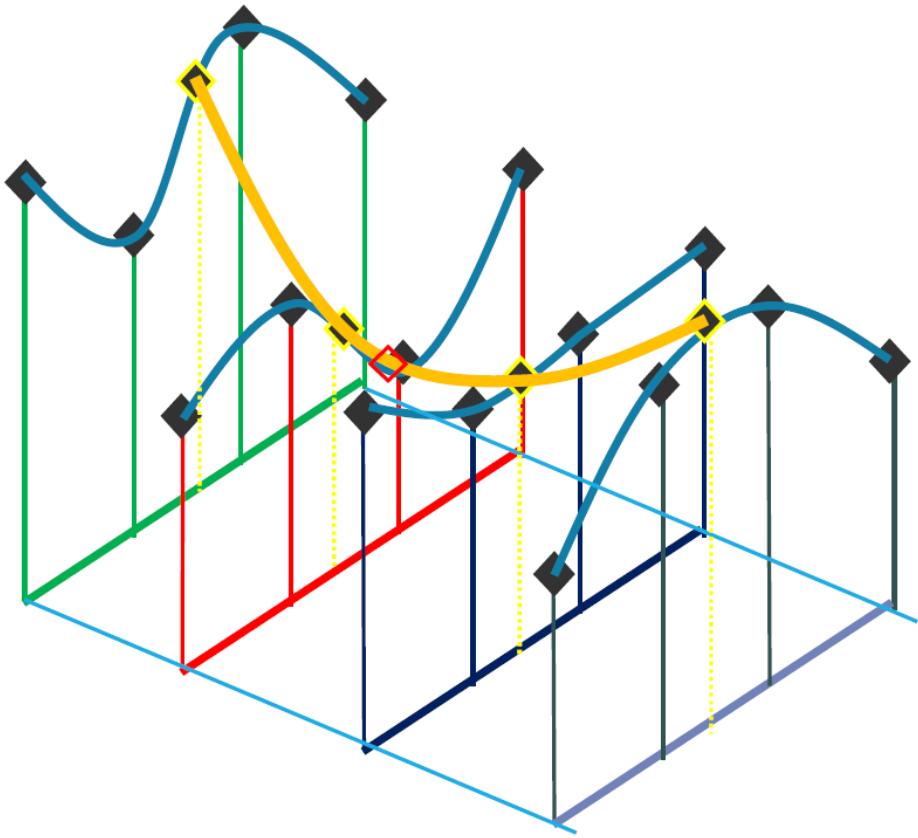


Abbildung 2.1: Auszug aus Folie 10.

c) Checkerboard und Laufzeiten

Die Checkerboard Funktion wird aus Übung 1 wiederverwendet.

Bei der Nearest Neighbour Implementierung werden die Pixel des Bildes einmal besucht und somit erhält man bei $n=\text{Anzahl der Pixel}$: $\mathcal{O}(n)$. Der Zugriff auf die Nachbarn erfolgt direkt mit $\mathcal{O}(1)$ und hat deshalb keinen Einfluss auf die Laufzeitberechnung.

Gleich verhält es sich bei der Bilinearen Interpolation, allerdings mit deutlich mehr Berechnungen und damit erhöhter CPU-Leistung.

Zu den Qualitäten siehe folgende Test.

Tests



Abbildung 2.2: 2x2 Checkerboard Ausgabe, auf den ersten Blick kaum erkennbar.

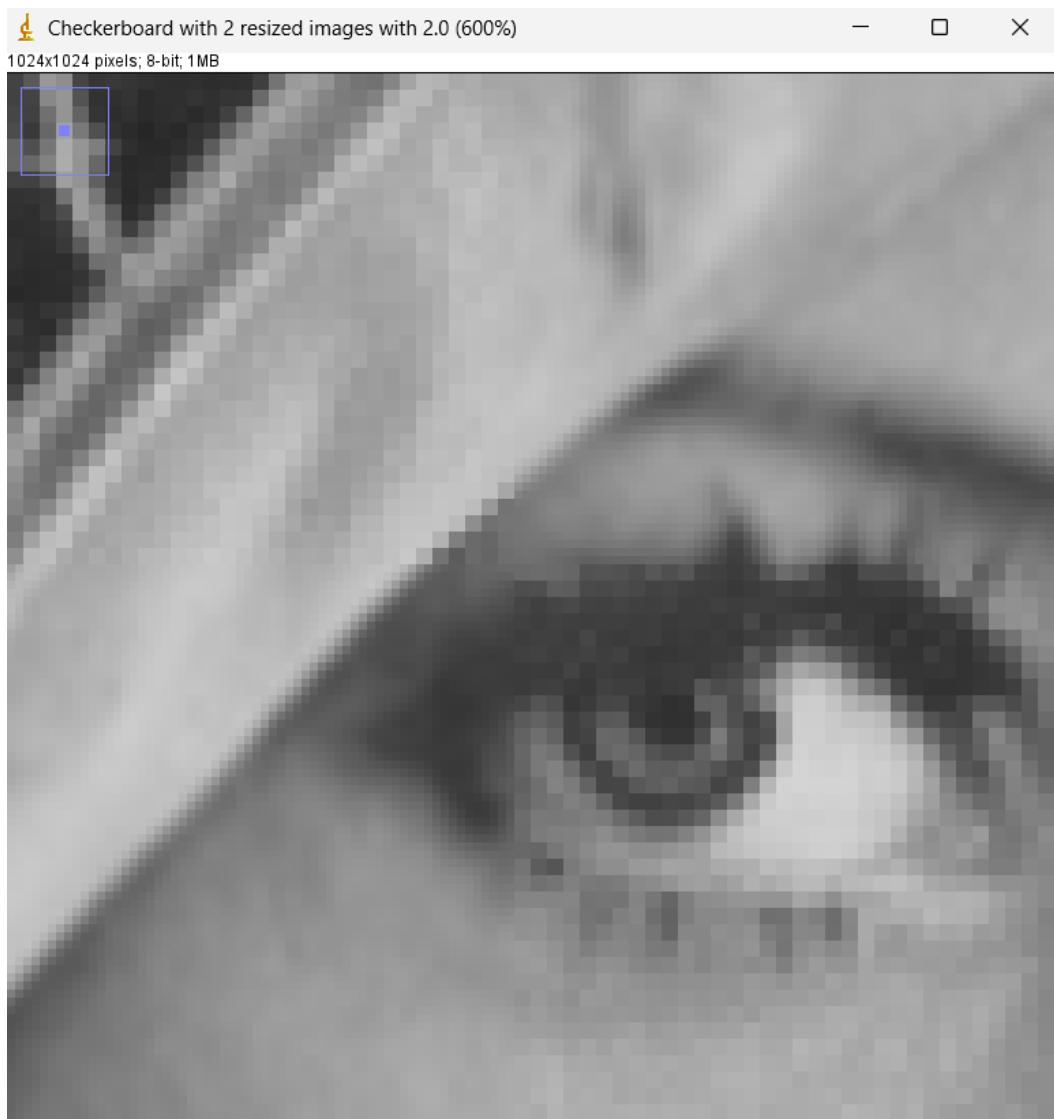


Abbildung 2.3: Bei näherem Hinsehen leistet die Bilineare Interpolation viel feinere Strukturen.



Abbildung 2.4: Differenzenbild von Lena. Schwarze Bereiche liefern das gleiche Ergebnis beider Interpolationen. Weiß zeigt die Unterschiede, besonders in detailreichen Bereichen.



Abbildung 2.5: 2x2 Checkerboard Ausgabe von Bridge, hineingezoomt. Bilineare Interpolation liefert einen hohen Detailgrad.

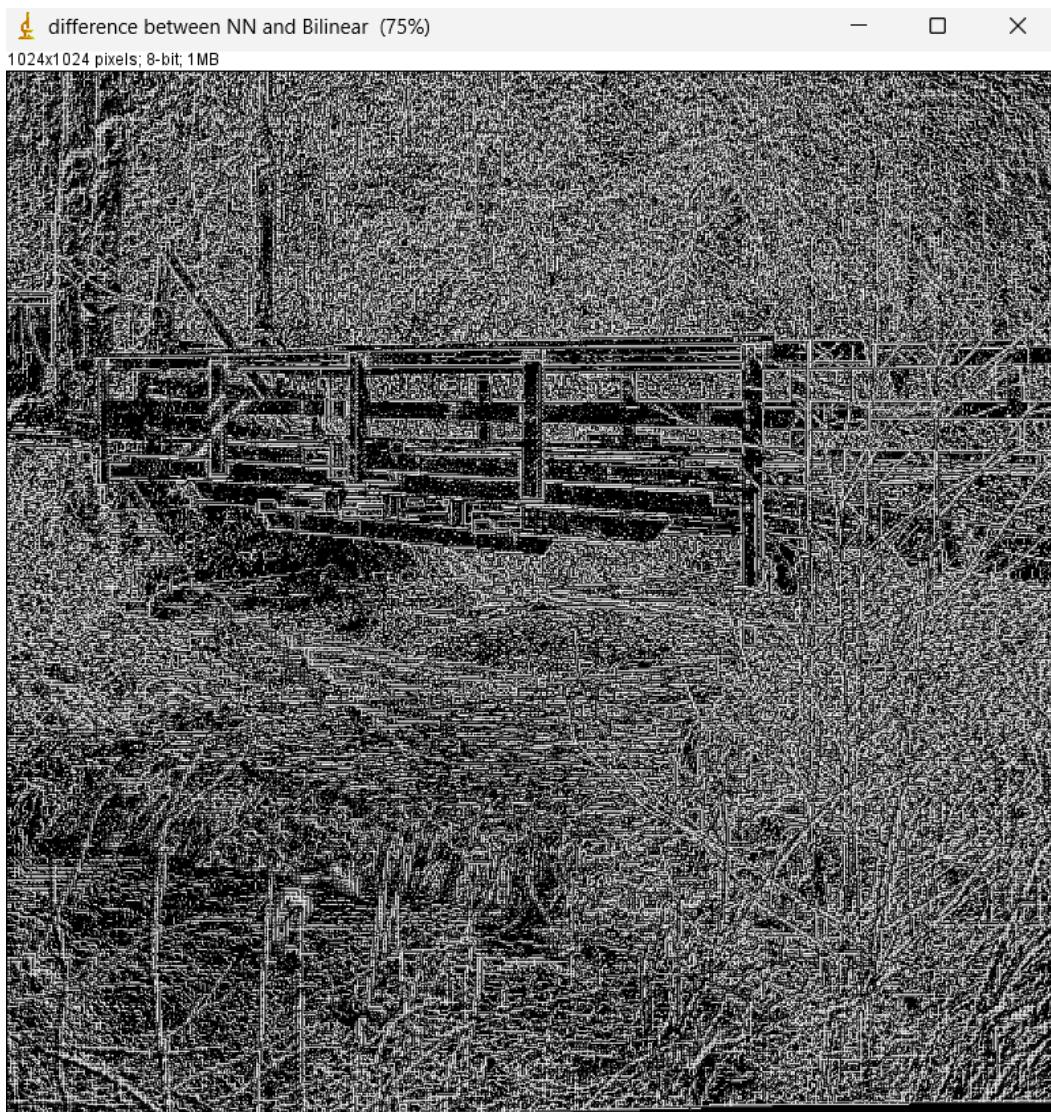


Abbildung 2.6: Differenzenbild von Bridge. Viel Weiß deutet auf stark unterschiedliche Ergebnispixel der beiden Interpolationen.



Abbildung 2.7: 2x2 Checkerboard Ausgabe von einem Baum, hineingezoomt. Wieder viele Details bei Bilinearer Interpolation.



Abbildung 2.8: 2x2 Checkerboard Ausgabe einer Mohnblume, hineingezoomt. Bei großflächig momtonen Bereichen erkennt man das Checkerboard kaum.



Abbildung 2.9: Trotzdem sind auch bei der Mohnblume viele Differenzen vorhanden.

2.2 Klassifizierung mittels Kompression

Die Lempel-Ziv Methode zur Komprimierung von Daten eignet sich unter anderem zum Klassifizieren verschiedener Daten. Dies stellt eine verhältnismäßig einfache und doch sehr beeindruckende Methode zur Klassifizierung dar. Da bei der Lempel-Ziv Methode sogenannte Wörterbücher erstellt werden, ist es möglich die Dateigröße einer Referenzdatei mit der Dateigröße der zu klassifizierenden Datei + Referenzdatei miteinander zu vergleichen. Ist die zu klassifizierende Datei ident mit der Referenzdatei, so werden keine weiteren Einträge dem Wörterbuch hinzugefügt. Ist die Datei jedoch völlig unterschiedlich, müssen sehr viele neue Einträge erstellt werden. Macht man sich das zu nutzen, lassen sich einfache Klassifizierungen händisch durchführen.

a) Klassifizieren von Sprachen

Zur Klassifikation der Sprache wurden repräsentative Vergleichsdatensätze in 8 verschiedenen Sprachen angelegt. Dabei handelt es sich um willkürlich erstellte Texte mit etwa 200 Wörtern. Wichtig dabei ist darauf zu achten, dass es sich um reine Textdateien (.txt) handelt. Ein anderes Dateiformat mit Informationen zur Formatierung könnte das Ergebnis verfälschen. Bei den zu klassifizierenden Texten wurden kurze Auszüge aus verschiedenen Wikipedia-Artikeln herangezogen. Diese werden anschließend in jedes .txt-File der Vergleichsdatensätze hineinkopiert und diese anschließend gezippt. Das stellt sicher, dass ein gemeinsames Wörterbuch angelegt wird. Vergleicht man nun die Größe des Vergleichsdatensatz (Reference) mit der Größe des zu klassifizierenden Text und Vergleichsdatensatz gemeinsam (Reference+Example), so erhält man die Differenz. Diese Differenz ist entsprechend kleiner, wenn die Sprache übereinstimmt.

References	Example 1		Example 2		Example 3		Example 4		Example 5	
	Byte	Byte	Difference Byte	Byte	Difference Byte	Byte	Difference Byte	Byte	Difference	
English	1 373	2 020	647	1 427	54	1 561	188	1740	367	1512
Romanian	1 517	2 017	500	1 421	-96	1 692	175	1893	376	1621
Spanish	1 455	1 968	513	1 368	-87	1 643	188	1837	382	1589
France	1 482	1 989	507	1 399	-83	1 648	166	1858	376	1616
Czech	1 484	1 996	512	1 403	-81	1 683	199	1884	400	1632
Italian	1 421	1 934	513	1 335	-86	1 603	182	1802	381	1559
Turkish	1 454	1 921	467	1 374	-80	1 648	194	1845	391	1600
Hungarian	1 518	2 024	506	1 441	-77	1 712	194	1915	397	1666
										148

Abbildung 2.10: Byte-Tabelle der Textdateien von Reference und zusammengeführten zip Dateien.

	Example 1 (Turkish)		Example 2 (Spanish)		Example 3 (France)		Example 4 (English)		Example 5 (Romanian)	
1	Turkish	467	1	Romanian	-75	1	French	166	1	English
2	Romanian	500	2	Spanish	-74	2	Romanian	175	2	Romanian
3	Hungarian	506	3	Italian	-70	3	Italian	182	3	French
	France	507		France	-68		English	188		Italian
	Czech	512		English	-67		Spanish	188		Spanish
	Spanish	513		Turkish	-64		Turkish	194		French
	Italian	513		Hungarian	-61		Hungarian	194		English
	English	647		Czech	-58		Czech	199		Turkish
										134
										146
										148

Abbildung 2.11: Ranking aus den Vergleichen der Sprachdateien.

Anhand der Konfusionsmatrix (siehe Abb. 2.12) wird ersichtlich, dass hinsichtlich des Best-Match fast jede Sprache identifiziert werden konnte. Bei dem Example 2 handelt es sich um einen sehr kurzen Text. Auffallend ist, dass die Klassifizierung bei sehr kurzen Texten weniger gut funktioniert als bei längeren Texten. So ist bei Example 2 kein signifikanter Unterschied erkennbar, während bei einem langen Text (Example 5) ein signifikanter Unterschied erkennbar ist. Gesamt konnte eine Trefferquote von 80% erzielt werden.

Example	Predicted	Reality	True/False	
Example 1	Turkish	Turkish	✓	1
Example 2	Romanian	Spanish	✗	0
Example 3	France	France	✓	1
Example 4	English	English	✓	1
Example 5	Romanian	Romanian	✓	1

Abbildung 2.12: Konfusionsmatrix.

b) OPTIONAL – nur für Interessierte/Expert*innen

Zur Klassifikation von Bildern wurden Vergleichsdatensätze von 4 verschiedenen Objekten (House, Tree, Wave, Bridge) angelegt. Für jedes Objekt wurden 6 verschiedene Bilder vorbereitet. Bei der Vorbereitung dieser gilt es einiges zu beachten, um eine Klassifizierung durchführen zu können. Um eine gleiche Pixelgröße und Farbtreue zu garantieren, wurden Screenshots in der Größe von 330x330 Pixel erstellt. Diese wurden anschließend in 8-bit Graustufenbilder umgewandelt. Durch die geringe Größe der Bilder wird außerdem sichergestellt, dass das Wörterbuch bei der Lempel-Ziv Methode nicht voll wird, sondern alles in einem Wörterbuch Platz findet. Zudem ist es wichtig ein nicht-komprimierendes Dateiformat zu wählen. Gewählt wurde das Dateiformat .png. Dasselbe Vorgehen wurde auch bei den Beispielbildern angewandt. Ähnlich wie bei der Sprach-Klassifizierung wurden auch hier die Beispielbilder gemeinsam mit den Bildern der Vergleichsdatensätze in eine gemeinsame Datei kopiert und anschließend gezippt.

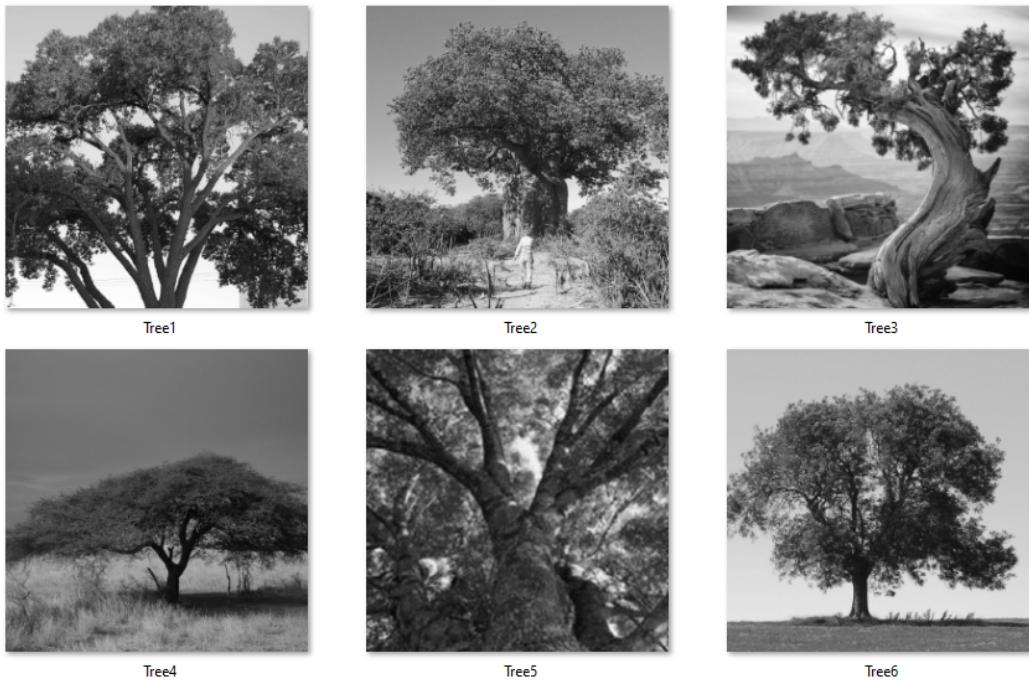


Abbildung 2.13: Reference Tree.



Abbildung 2.14: Reference House.

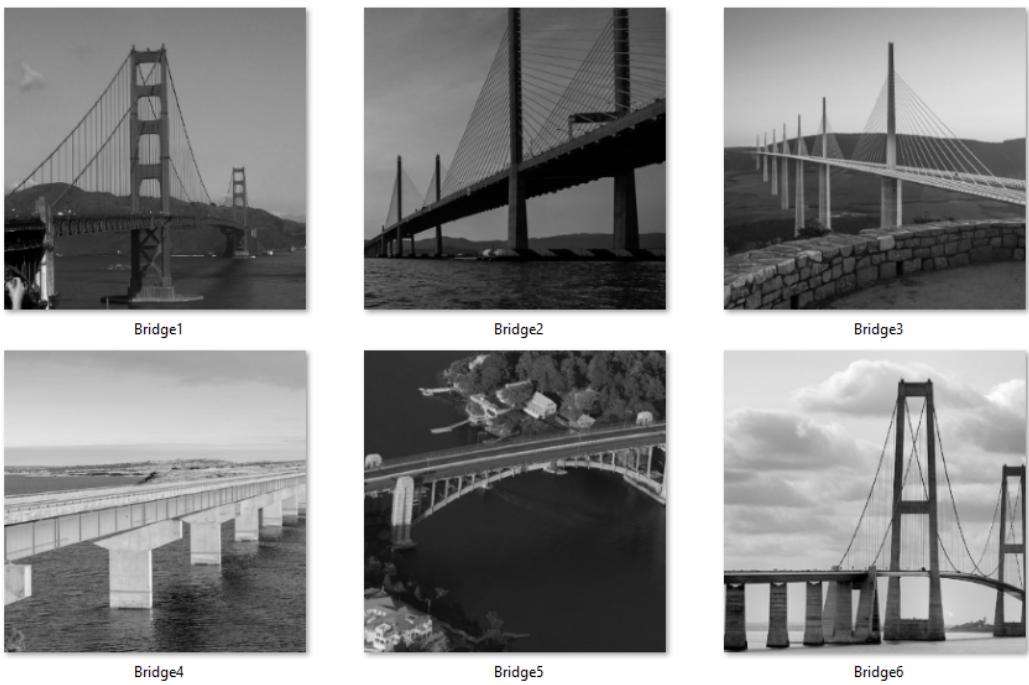


Abbildung 2.15: Reference Bridge.



Abbildung 2.16: Reference Wave.

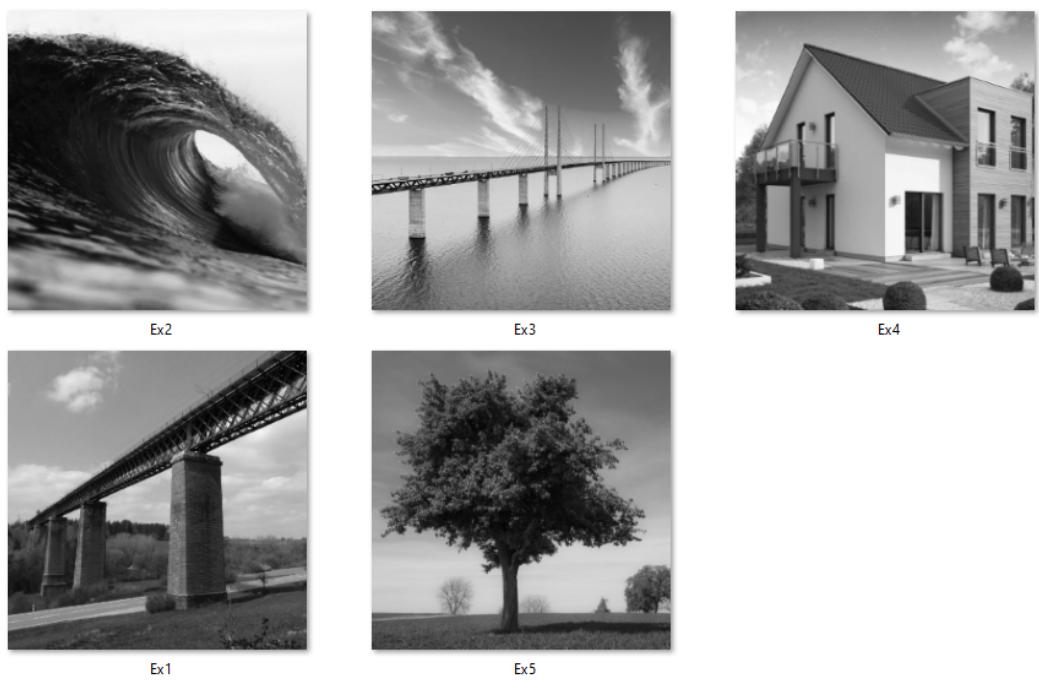


Abbildung 2.17: Example Bilder.

References	Example 1 (Bridge)		Example 2 (Wave)		Example 3 (Bridge)		Example 4 (House)		Example 5 (Tree)		
Byte	Byte	Difference	Byte	Difference	Byte	Difference	Byte	Difference	Byte	Difference	
Tree 1	88 400	167 555	79 155	172 033	83 633	165 381	76 981	166 994	78 594	165 662	77 262
Tree 2	92 679	171 792	79 113	176 015	83 336	169 593	76 914	171 434	78 755	170 030	77 351
Tree 3	92 147	171 298	79 151	175 681	83 534	169 513	77 366	170 629	78 482	169 524	77 377
Tree 4	75 846	154 733	78 887	159 024	83 178	152 907	77 061	154 643	78 797	153 057	77 211
Tree 5	97 373	176 074	78 701	180 860	83 487	174 303	76 930	175 745	78 372	174 431	77 058
Tree 6	74 307	153 178	78 871	157 830	83 523	151 276	76 969	152 665	78 358	151 466	77 159
House 1	74 713	154 536	79 823	158 181	83 468	152 149	77 436	153 512	78 799	152 933	78 220
House 2	67 009	146 658	79 649	150 652	83 643	144 292	77 283	145 845	78 836	144 897	77 888
House 3	80 437	159 652	79 215	163 808	83 371	157 666	77 229	158 952	78 515	157 862	77 425
House 4	78 112	157 196	79 084	161 647	83 535	155 288	77 176	156 754	78 642	155 397	77 285
House 5	85 014	164 398	79 384	168 828	83 814	162 538	77 524	163 747	78 733	162 813	77 799
House 6	71 593	151 446	79 853	154 828	83 235	149 478	77 885	150 551	78 958	149 998	78 405
Bridge 1	69 636	149 138	79 502	153 501	83 865	146 928	77 292	148 888	79 252	147 471	77 835
Bridge 2	67 280	146 242	78 962	150 739	83 459	144 466	77 186	146 142	78 862	144 558	77 278
Bridge 3	65 671	144 573	78 902	148 651	82 980	142 611	76 940	144 035	78 364	142 773	77 102
Bridge 4	68 391	147 935	79 544	151 807	83 416	145 543	77 152	147 137	78 746	146 325	77 934
Bridge 5	63 411	142 084	78 673	147 408	83 997	141 033	77 622	141 997	78 586	140 539	77 128
Bridge 6	70 723	150 142	79 419	154 532	83 809	148 644	77 921	149 461	78 738	148 731	78 008
Wave 1	81 419	160 329	78 910	164 664	83 245	158 602	77 183	160 092	78 673	158 415	76 996
Wave 2	95 224	174 627	79 403	178 891	83 667	172 997	77 773	173 818	78 594	173 125	77 901
Wave 3	69 231	149 326	80 095	153 269	84 038	147 400	78 169	148 405	79 174	147 815	78 584
Wave 4	74 227	153 726	79 499	157 786	83 559	151 743	77 516	153 030	78 803	151 913	77 686
Wave 5	81 398	160 940	79 542	164 148	82 750	158 412	77 014	160 031	78 633	159 237	77 839
Wave 6	60 962	139 794	78 832	143 924	82 962	137 938	76 976	139 569	78 607	138 029	77 067

Abbildung 2.18: Byte-Tabelle der Bilddateien von Reference und zusammengeführten zip Dateien.

Example 1 (Bridge)		Example 2 (Wave)		Example 3 (Bridge)		Example 4 (House)		Example 5 (House)			
Rank	Rank	Rank	Rank	Rank	Rank	Rank	Rank	Rank	Rank		
1 Bridge 5	78 673	1	Wave 5	82 750	1	Tree 2	76 914	1	Tree 6	78 358	1
2 Tree 5	78 701	2	Wave 6	82 962	2	Tree 5	76 930	2	Bridge 3	78 364	2
3 Wave 6	78 832	3	Bridge 3	82 980	3	Bridge 3	76 940	3	Tree 5	78 372	3
Tree 6	78 871	Tree 4	83 178	Tree 6	76 969	Tree 3	78 482	Tree 3	77 102	Bridge 3	77 102
Tree 4	78 887	House 6	83 235	Wave 6	76 976	House 3	78 515	House 3	77 128	Bridge 5	77 128
Bridge 3	78 902	Wave 1	83 245	Tree 1	76 981	Bridge 5	78 586	Tree 6	77 159	Tree 6	77 159
Wave 1	78 910	Tree 2	83 336	Wave 5	77 014	Tree 1	78 594	Tree 4	77 211	Tree 4	77 211
Bridge 2	78 962	House 3	83 371	Tree 4	77 061	Wave 2	78 594	Wave 2	77 262	Bridge 2	77 262
House 4	79 084	Bridge 4	83 416	Bridge 4	77 152	Wave 6	78 607	Wave 6	77 278	House 4	77 285
Tree 2	79 113	Bridge 2	83 459	House 4	77 176	Wave 5	78 633	House 4	77 285	Tree 2	77 351
Tree 3	79 151	House 1	83 468	Wave 1	77 183	House 4	78 642	Tree 2	77 351	Tree 3	77 377
Tree 1	79 155	Tree 5	83 487	Bridge 2	77 186	Wave 1	78 673	Tree 3	77 377	House 3	77 425
House 3	79 215	Tree 6	83 523	House 3	77 229	House 5	78 733	House 3	77 425	House 5	77 425
House 5	79 384	Tree 3	83 534	House 2	77 283	Bridge 6	78 738	Wave 4	77 686	Bridge 6	77 686
Wave 2	79 403	House 4	83 535	Bridge 1	77 292	Bridge 4	78 746	Bridge 4	77 799	House 5	77 799
Bridge 6	79 419	Wave 4	83 559	Tree 3	77 366	Tree 2	78 755	Bridge 1	77 835	Bridge 1	77 835
Wave 4	79 499	Tree 1	83 633	House 1	77 436	Tree 4	78 797	Wave 5	77 839	Wave 5	77 839
Bridge 1	79 502	House 2	83 643	Wave 4	77 516	House 1	78 799	House 2	77 888	House 2	77 888
Wave 5	79 542	Wave 2	83 667	House 5	77 524	Wave 4	78 803	Wave 2	77 901	Wave 2	77 901
Bridge 4	79 544	Bridge 6	83 809	Bridge 5	77 622	House 2	78 836	Bridge 4	77 934	Bridge 4	77 934
House 2	79 649	House 5	83 814	Wave 2	77 773	Bridge 2	78 862	Bridge 6	78 008	Bridge 6	78 008
House 1	79 823	Bridge 1	83 865	House 6	77 885	House 6	78 898	House 1	78 220	House 1	78 220
House 6	79 853	Bridge 5	83 997	Bridge 6	77 921	Wave 3	79 174	House 6	78 405	House 6	78 405
Wave 3	80 095	Wave 3	84 038	Wave 3	78 169	Bridge 1	79 252	Wave 3	78 584	Wave 3	78 584

Abbildung 2.19: Ranking aus den Vergleichen der Bilddateien.

Anhand der Tabelle (Abb. 2.18) und dem Ranking (Abb. 2.19) ist ersichtlich, dass die Klassifikation hierbei eine geringere Trefferquote aufweist, als bei den Sprachen. Grund hierfür können viele Faktoren sein, wie etwa abweichende Graustufen. Über alle Beispiele betrachtet, lässt sich kein signifikanter Unterschied zu einer rein zufälligen Auswahl feststellen. Um dies zu erreichen, wäre es notwendig die Bilder besser abzugleichen.

2.3 Kompression und Code-Transformation

a) Lempel-Ziv Kompression

Bei der Lempel-Ziv Kompression wird über die Zeichenkette iteriert, und jeweils ein aktueller Wert und der nächste Wert gespeichert. Wird die Konkatenation **C** aus aktuellem und nächstem Wert nicht im Wörterbuch gefunden, dann wird **C** im Wörterbuch abgespeichert und der aktuelle Wert ausgegeben. Wird **C** jedoch im Wörterbuch gefunden, dann wird **C** zum aktuellen Wert und es wird wie zuvor beschrieben weitergesucht. Aus ursprünglich 25 Zeichen werden 16 Zeichen. Das ergibt eine Kompressionsrate von $\frac{25}{16} = 1,5625$. Das Wörterbuch muss dazu nicht übertragen werden, der Empfänger kann das aus dem Out Ergebnis zurückrechnen.

Tabelle 2.2: Lempel-Ziv Kompression der Sequenz **dabbababbbbbaaaabababccdd**

<i>Actual</i>	<i>Next</i>	<i>Found</i>	<i>Add</i>	<i>Out</i>
d	a	da-N	da:256	d(100)
a	b	ab-N	ab:257	a(97)
b	b	bb-N	bb:258	b(98)
b	a	ba-N	ba:259	b(98)
a	b	ab-Y		
ab	a	aba-N	aba:260	ab(257)
a	b	ab-Y		
ab	a	aba-Y		
aba	b	abab-N	abab:261	aba(260)
b	b	bb-Y		
bb	b	bbb-N	bbb:262	bb(258)
b	a	ba-Y		
ba	a	baa-N	baa:263	ba(259)
a	a	aa-N	aa:264	a(97)
a	a	aa-Y		
aa	b	aab-N	aab:265	aa(264)
b	a	ba-Y		
ba	b	bab-N	bab:266	ba(259)
b	a	ba-Y		
ba	b	bab-Y		
bab	c	babc-N	babc:267	bab(266)
c	c	cc-N	cc:268	c(99)
c	d	cd-N	cd:269	c(99)
d	d	dd-N	dd:270	d(100)
d				d(100)

b) Huffman Coding

Beim Huffman Coding verwenden wir eine etwas andere Vorgehensweise als wir in der Vorlesung besprochen haben (siehe Abb. 2.20). Die Vorkommen der einzelnen Zeichen werden kummuliert und danach aufsteigend sortiert. Für den Huffmann Baum werden die beiden Zeichen mit dem geringsten Vorkommen summiert und als neuer Elternknoten gespeichert. Die Kanten werden dazu von links nach rechts mit 0 und 1 beschriftet. Dieser Knoten und das nächste Zeichen in der aufsteigenden Sortierung ergeben zusammen wiederum einen neuen Elternknoten und die Kanten wie zuvor beschriftet. Das

wird solange fortgeführt, bis alle vorhandenen Knoten verbunden sind. Das Ergebnis ist ein Binärbaum, dessen Kantenpfade von der Wurzel bis zum Zeichen den dazugehörigen Binär-Code liefert.

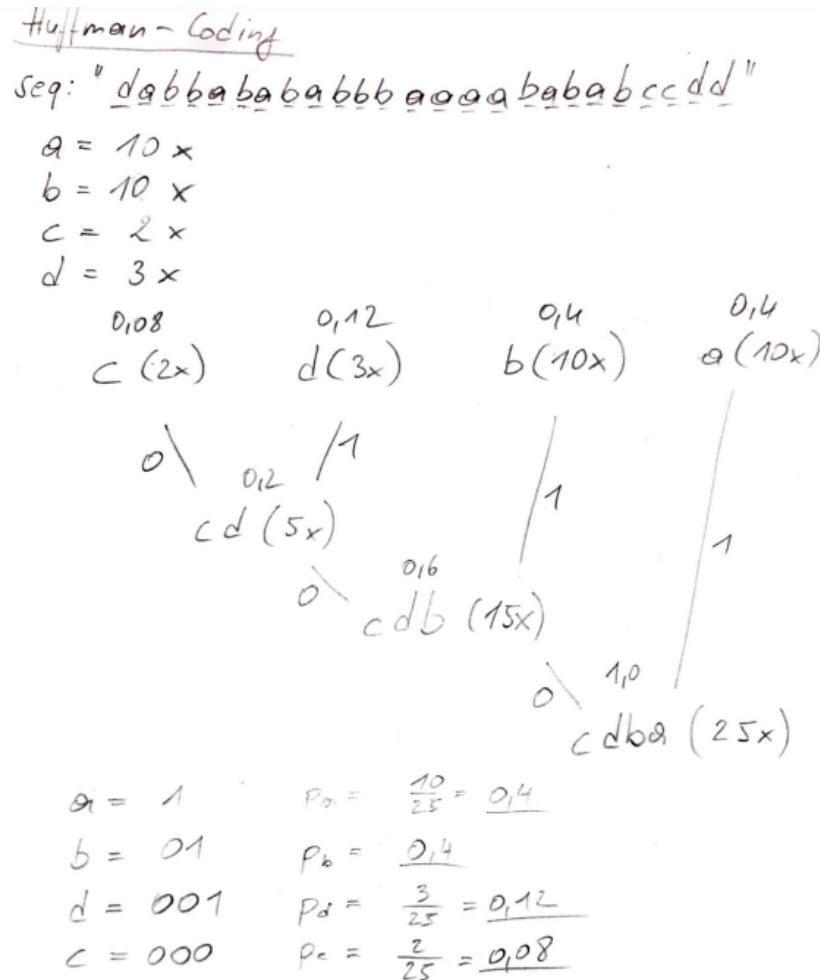


Abbildung 2.20: Huffman Baum der Sequenz dabbabababbbaaaaabababccdd

Mittlere Codewortlänge und Kompressionsrate L

Das Zeichen **a** wird mit einem Bit abgespeichert, **b** mit 2 Bit und **c** sowie **d** mit 3 Bit. Multipliziert mit dem Vorkommen der Wahrscheinlichkeit jedes Zeichens ergibt das eine Kompressionsrate von: $L = 1 \cdot 0,4 + 2 \cdot 0,4 + 3 \cdot 0,12 + 2 \cdot 0,08 = 1,72$.

Maximale Kompressionsrate

Für die Aufgabenstellung der minimalen und maximalen Kompressionsrate einer 10-stelligen Sequenz werden die beiden Extremfälle untersucht. Erster Fall, alle Zeichen sind unterschiedlich **Sequenz: abcdefghij**. Ein Zeichensatz von 10 unterschiedlichen

Zeichen benötigt vor der Kompression einen Datentyp mit einem Wertebereich von $2^4 = 16$. Die Wahrscheinlichkeit für jedes Zeichen sind $\frac{1}{10}$. Es ergibt sich eine mittlere Codewortlänge von $L = \frac{6 \cdot 3 + 4 \cdot 4}{10} = 3,4$ (siehe Abb. 2.21). Und eine *Kompressionsrate* = $\frac{10 \cdot 2^4}{10 \cdot 3,4} \approx 4,71$

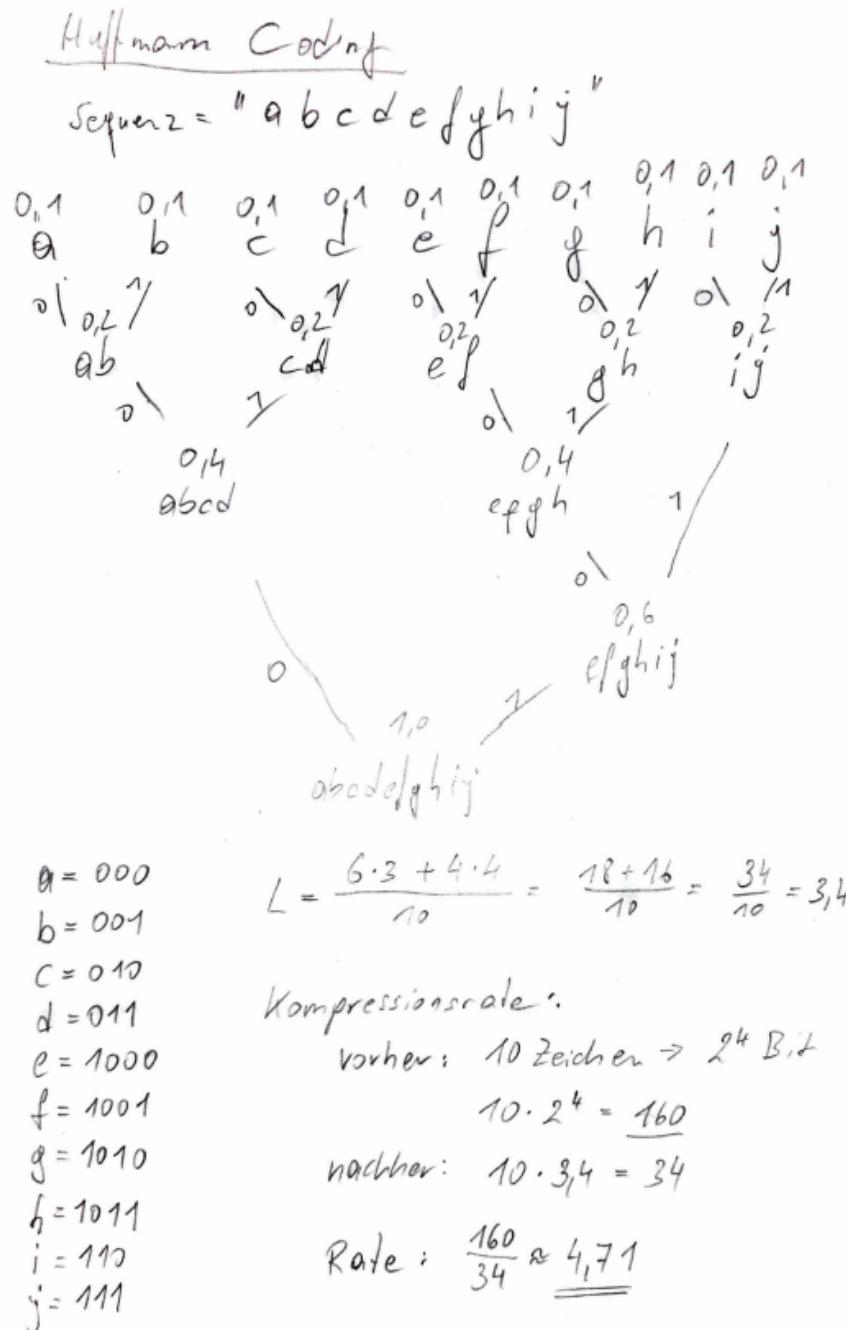


Abbildung 2.21: Huffman Coding mit der 10-stelligen Sequenz abcdefghij

Minimale Kompressionsrate

Der andere Extremfall ist, dass 10-mal das gleiche Zeichen Sequenz: **aaaaaaaaaa**. Das einzige Codewort das hier nötig ist, ist **a=1** und benötigt sowohl vor als auch nach der Kompression 1-Bit. Mittlere Codewortlänge $L = 1$. $Kompressionsrate = \frac{10 \cdot 1}{10 \cdot 1} = 1$. Hier wird nichts komprimiert.

c) Runlength Coding

Es gilt die Sequenz **11110101011110000011110001010** mit den Symbolen 0, 1 zu codieren. Beim Runlength Coding Verfahren werden die direkt aufeinander folgenden Vorkommen der gleichen Symbole gezählt und als Wert gespeichert. Es ergibt sich die neue Sequenz **41111155431111** mit 14 Stellen. Betrachtet man rein die Länge der Sequenzen, errechnet sich daraus eine Kompressionsrate von $\frac{30}{14} \approx \frac{1}{2}$. Betrachtet man aber auch den Speicher, der zur Persistierung der Werte nötig ist, dann ist die Kompressionsrate $\frac{30}{14 \cdot 2^3} \approx 0,27$ und damit ist die neue Datei fast 4x so groß wie zuvor. Denn der gespeicherte Wertebereich besteht aus 1, 2, 3, 4, 5 und um 5 verschiedene Werte speichern zu können, benötigt man 2^3 Bit pro Zeichen. Generell ist dieses Verfahren stark von der Homogenität der Quellsequenz abhängig.

Erweiterung der Zeichen bei Runlength Coding

Da man bei einer Erweiterung des Wertebereichs während des Kodierens nicht mehr automatisch zwischen den binären Zeichen hin- und herschalten kann, muss man einen Weg finden das momentan gezählte Symbol zu identifizieren. Es bietet sich an das Symbol einmal in die Ergebnissequenz zu schreiben, gefolgt von der Anzahl des Vorkommens bis zum nächsten Symbol (siehe Abb. 2.22). Da hier ein weiteres Zeichen pro Symbolwechsel in die Zielsequenz geschrieben wird, ist es hier noch deutlicher ersichtlich, dass die Quelle nicht stark inhomogen sein darf. Verglichen zur binären Quelle wird hier das Ziel mindestens doppelt so groß. Es darf auch keine Schnittmenge von gelesenen Symbolen zu Symbolen, die zum Zählen verwendet werden, geben.

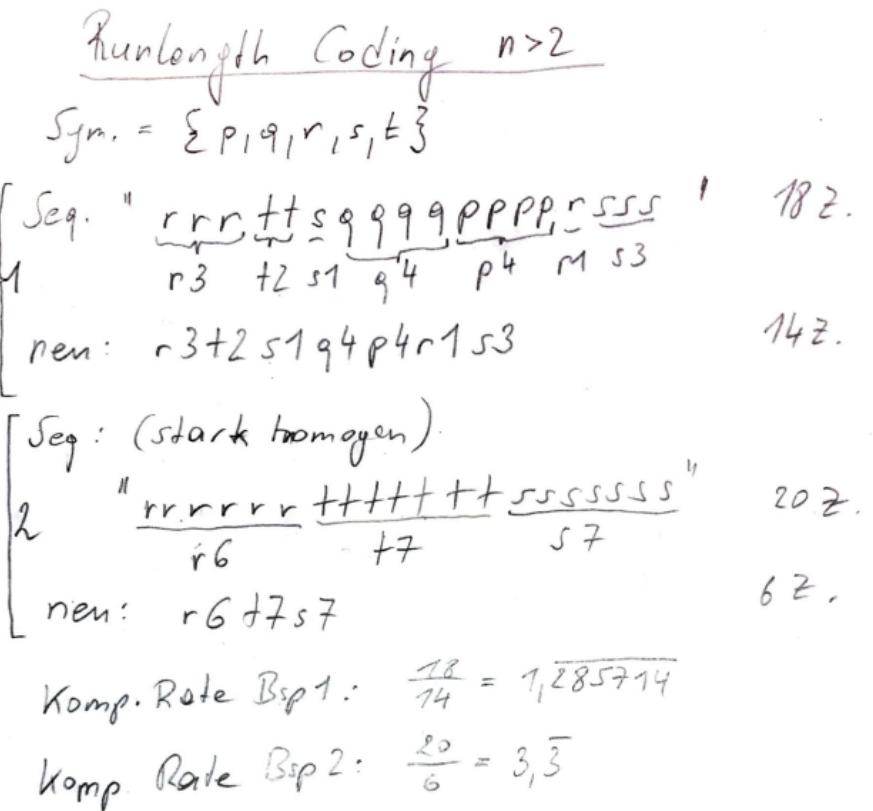


Abbildung 2.22: Runlength Coding mit erweitertem Quellzeichen.

d) Berechnung der Entropie

Es gilt die 30-stellige Sequenz **221111226611122333345645112111** auf Entropie zu untersuchen (siehe Abb. 2.23). Die Entropie beschreibt den Informationsgehalt einer Zeichenkette. Dazu werden die Wahrscheinlichkeiten der vorhandenen Zeichen erfasst und diese dann gemäß Gleichung 2.9 berechnet.

$$\text{Entropie} = - \sum_{i=1}^n p_i \cdot \log_2 p_i \quad (2.9)$$

Entropie

22 1111 22 66 111 22 3333 456 45112111

$$1: 12x \quad p_1 = \frac{12}{30} = 0,4$$

$$2: 7x \quad p_2 = \frac{7}{30} = 0,2\bar{3}$$

$$3: 4x \quad p_3 = \frac{4}{30} = 0,1\bar{3}$$

$$4: 2x \quad p_4 = \frac{2}{30} = 0,0\bar{6}$$

$$5: 2x \quad p_5 = \frac{2}{30} = 0,0\bar{6}$$

$$6: 3x \quad p_6 = \frac{3}{30} = 0,1$$

$$-\sum_{i=1}^n p_i \cdot \log_2(p_i) = -(0,4 \cdot \log_2(0,4) + 0,2\bar{3} \cdot \log_2(0,2\bar{3}) + 0,1\bar{3} \cdot \log_2(0,1\bar{3}) + 2 \cdot [0,0\bar{6} \cdot \log_2(0,0\bar{6})] + 0,1 \cdot \log_2(0,1)) = \underline{\underline{2,25932}}$$

Abbildung 2.23: Berechnung der Entropie.

Minimale Entropie

Analog zur minimalen Kompressionsrate wird dazu die Sequenz **aaaaaaaaaa** gewählt. Die Wahrscheinlichkeit für das Symbol **a** ist 1 und in die Gleichung 2.9 eingesetzt ergibt sich $-1 \cdot \log_2 1 = 0$

Maximale Entropie

Dazu wird wiederum die Sequenz **abcdefghijkl** herangezogen. Die Wahrscheinlichkeit für jedes Zeichen ist $\frac{1}{10} = 0,1$. Daraus folgt $-0,1 \cdot \log_2 0,1 \approx 3,32$

Entropie und Kompression

Es wird vermutet wenn zur Kompression die Wahrscheinlichkeiten der Symbole verwendet wird, dann ist die Entropie ausschlaggebend, siehe Huffman Coding. Es gibt aber auch Kompressionsverfahren die nicht auf Wahrscheinlichkeiten beruhen, wie die Run-length Coding Verfahren. Implizit hat die Entropie zwar Einfluss, denn mit Erweiterung der Symbolwerte, wird auch die Quelle inhomogener. Aber die Frage lautet *ist lediglich die Auftrittswahrscheinlichkeit entscheidend*, das könnte man mit Nein beantworten.