Zum erfolgreichen Absolvieren der Lehrveranstaltung „Ausgewählte Themen der Medieninformatik“ ist die Lösung für die auf den folgenden Seiten beschriebene Hausarbeit einzureichen.

Bitte beachten Sie dabei die folgenden Hinweise:

(i) Sie müssen sich in HELIOS für diese Prüfung angemeldet haben.

(ii) Ihre Lösungen müssen bis zum Abend des 13.07.2016 als ZIP-Archiv per E-Mail bei mir eingegangen sein.

(iii) Reichen Sie pro Aufgabe eine `.py`-Datei ein, in der der vollständige Code zur Lösung enthalten ist; **keine** JUPYTER-Notebooks. Die Datei soll so aufgebaut sein, dass ich sie in SPYDER laden und direkt ausführen kann. (Insbesondere sollten keine Dateipfade, die nur auf Ihrem Computer existieren, fest eincodiert sein!)

(iv) Schreiben Sie Code, der mit der aktuellen Version von PYTHON 3 funktioniert.

(v) Verwenden Sie „sprechende Namen“ (statt `a` oder `f`) für Variablen und Funktionen und kommentieren Sie Ihren Code, wenn nicht offensichtlich ist, was er tut.

(vi) Sie können in Teams arbeiten. Ein Team darf aus maximal drei Personen bestehen. Die Mitglieder des Teams sollten selbstverständlich bei der Abgabe der Lösung alle genannt werden.

(vii) Sie müssen mindestens drei Aufgaben vollständig gelöst haben. Für eine sehr gute Note müssen Sie *alle* Aufgaben gelöst haben.

(viii) Sie dürfen, wenn in der jeweiligen Aufgabe nichts anderes steht, sämtliche Bibliotheken der *Standard Library* sowie die, die in der Vorlesung behandelt wurden, benutzen. Die Benutzung anderer Bibliotheken ist **nicht** erlaubt.

(ix) Wenn Sie Code von Dritten (z.B. aus einem Artikel auf STACK OVERFLOW) übernehmen, dann weisen Sie im Inline-Kommentar darauf hin. Sollte ich bei Stichproben fremden Code entdecken, der nicht als solcher gekennzeichnet ist, muss ich das als Täuschungsversuch werten.

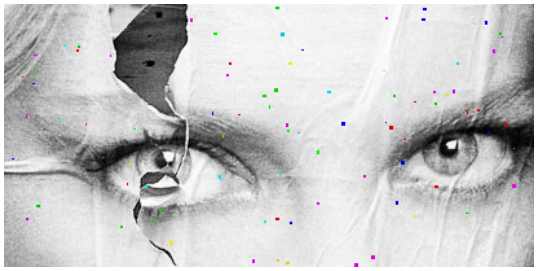
(x) Wenn Sie vorhaben, fremden Code in größerem Umfang zu nutzen, sprechen Sie das vorher mit mir per E-Mail ab. Die Aufgaben sind so gestaltet, dass Sie selbstständig in der Lage sein sollten, sie zu lösen.

(xi) Die Effizienz Ihres Codes spielt keine besonders große Rolle, die korrekte Lösung der Aufgabe natürlich schon. Für die Bewertung relevant ist außerdem, ob Ihr Code elegant, sauber strukturiert und gut zu verstehen ist. Hingegen führen z.B. unnötige Wiederholungen und „Spaghetti-Code“ ggf. zu Punktabzug.

(xii) Stellen Sie inhaltliche Fragen zu den Aufgaben über das EMIL-Forum.

(xiii) Im Falle von Unklarheiten behalte ich mir vor, das Team zu mir einzuladen und mir den eingereichten Code erklären zu lassen.

Aufgabe 1. Im Ordner zu dieser Aufgabe finden Sie diverse Graustufenbilder, die durch bunte Pixel verunreinigt wurden.



Schreiben Sie eine Funktion `clean`, die die Bilder (ohne Kenntnis der Originale) möglichst gut restauriert. Die Funktion soll also als Ergebnis ein Graustufenbild liefern, in dem die bunten Pixel durch „passende“ graue ersetzt wurden. (Es gibt dabei je Pixel nicht *den* richtigen Wert. Wichtig ist der optische Eindruck – das Ergebnis soll nachher keine störenden Artefakte mehr haben.)

Die Funktion soll so aufgerufen werden:

```
clean("c:/Users/tick/Desktop/dirty.png", "c:/tmp/clean.png")
```

Aufgabe 2. Im Ordner zu dieser Aufgabe finden Sie eine große Menge von CAPTCHAs, die in etwa so aussehen:

Sie sollen eine Funktion schreiben, die so ein Bild als Eingabe bekommt und als Ausgabe den entsprechend String, in diesem Fall also `"4242"`, zurückgibt. Gehen Sie dabei so vor:

(i) Entfernen Sie zunächst das „Hintergrundrauschen“.

(ii) Isolieren Sie dann die einzelnen Ziffern. (Sie können sich darauf verlassen, dass jedes CAPTCHA aus genau vier Dezimalziffern besteht, die sich nicht berühren.)

(iii) Benutzen Sie die Bilder der Ziffern als Eingabe für eine logistische Regression.

Schreiben Sie zwei Funktionen:

- `prepare` bekommt als Eingabe einen Ordner mit Trainingsbildern und gibt als Ausgabe ein Objekt Ihrer Wahl zurück.

- `crack` bekommt als Eingabe das obige Objekt sowie ein CAPTCHA und gibt den zugehörigen String aus.

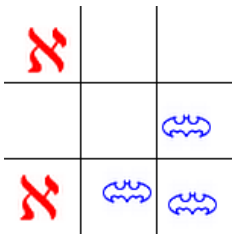
Es soll also so funktionieren:

```
something = prepare("/Users/trick/Desktop/CAPTCHAs/")
crack(something, "/Users/trick/Desktop/input.png")
```

Natürlich kann `crack` (im Gegensatz zu `prepare`) nicht davon ausgehen, dass der Name einer CAPTCHA-Datei das korrekte Ergebnis verrät.

Anmerkung: Es ist möglich, eine Trefferquote von 100% zu erreichen. Die Aufgabe gilt aber auch schon als gelöst, wenn Sie lediglich mehr als die Hälfte der CAPTCHAs richtig zuordnen.

Aufgabe 3. Sie sollen eine Funktion `solve` schreiben, die eine Stellung im Spiel *Tic-Tac-Toe* auflöst. Ihre Funktion bekommt als Eingabe ein Bild (siehe Ordner zu dieser Aufgabe) einer Spielsituation, die z.B. so aussehen kann:



Ausgeben soll sie als Tupel Zeile und Spalte des Feldes, in dem der Spieler, der am Zug ist, seinen Spielstein plazieren muss, um zu gewinnen. In diesem Fall müsste die Ausgabe also `(2,1)` sein, da der Spieler mit den roten Aleph-Zeichen dran ist und er gewinnt, wenn er einen Stein auf das erste Feld in der zweiten Zeile setzt.

- Sie können sich darauf verlassen, dass die Bilder immer dieselbe Größe haben und dass die schwarzen Markierungslinien zum Trennen der Felder immer exakt an derselben Stelle sind.
- Sie wissen vorab nicht, wie die „Spielsteine“ aussehen und wo genau sie auf dem Spielbrett sind. Sie liegen aber immer komplett innerhalb eines Feldes.
- Ihre Funktion bekommt als Eingabe garantiert nur Spielsituationen, in denen eindeutig ist, wer am Zug ist, und in denen es genau einen möglichen Gewinnzug gibt.

Aufgerufen werden soll die Funktion z.B. folgendermaßen:

```
solve("c:/tmp/TicTacToe/042.png")
```

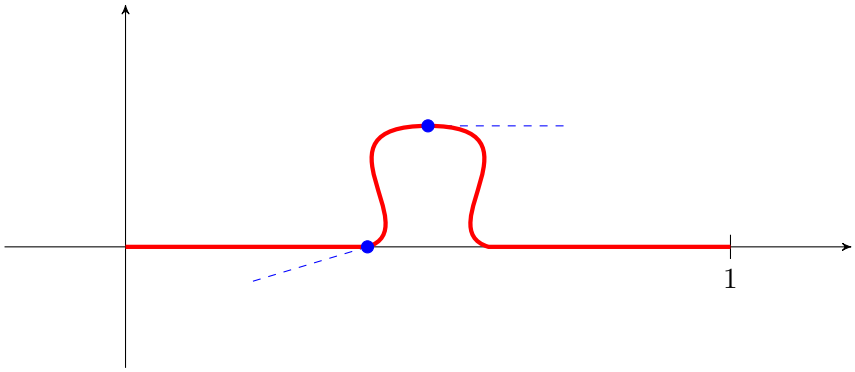
Aufgabe 4. Schreiben Sie eine Funktion `puzzle`, die als Eingabe ein (farbiges) Bild bekommt und als Ausgabe ein Puzzle wie dieses ausgibt. Die Funktion soll z.B. so aufgerufen werden:

```
puzzle("c:/Users/track/Desktop/input.jpg", "c:/tmp/puzzle.gif", 6, 7)
```

Damit soll das Bild `input.jpg` in das animierte GIF `puzzle.gif` mit sechs Reihen und sieben Spalten von Puzzlesteinen konvertiert werden. Sie können davon ausgehen, dass die Parameter für die Anzahl der Reihen und Spalten „vernünftige“ Werte haben, also weder extrem groß noch sehr klein sind. (Allerdings muss Ihr Programm auch dann funktionieren, wenn z.B. die Breite des Bildes in Pixeln nicht durch die Anzahl der Spalten teilbar ist.) Das Puzzle soll natürlich wie im Beispiel in zufälliger Reihenfolge zusammengesetzt werden.

Für diese Aufgabe darf **nicht** die Bibliothek MATPLOTLIB benutzt werden.

Hinweis: Lösen Sie die Aufgabe zunächst für rechteckige Puzzlestücke. Die typische Form der „richtigen“ Stücke kann man dann durch Bézierkurven hinbekommen:



Aufgabe 5. Sie haben den Code eines gegnerischen Geheimdienstes geknackt: Die feindlichen Agenten schicken sich gegenseitig ganze Zahlen. Diese werden als quadratische Matrizen verschickt, in denen nur Nullen und Einsen stehen, und die die Binärdarstellung der jeweiligen Zahl repräsentieren. Z.B. stehen

0	1	0	1
0	1	0	0
0	0	0	0
0	0	0	0

und

1	1	0
1	1	1
1	0	0

für $1 \cdot 2^1 + 1 \cdot 2^3 + 1 \cdot 2^5 = 42$ bzw. für $1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6 = 123$.

Die Matrizen wiederum werden als *Zeilen-* und *Spaltenspezifikationen* codiert. Z.B. würde

die erste Matrix von oben durch `[[1, 1], [1], [], []]` und `[[], [2], [], [1]]` dargestellt werden. Dabei steht jede Liste für eine Zeile bzw. Spalte (also etwa `[1, 1]` für die erste Zeile und `[2]` für die zweite Spalte) und jede Zahl in so einer Liste für einen *Block* von Einsen, d.h. für eine maximal lange Sequenz von direkt aufeinanderfolgenden Einsen. Die zweite Matrix würde also durch `[[2], [3], [1]]` und `[[3], [2], [1]]` codiert werden.

Hier noch ein Beispiel:

	`[2, 1]`	`[1]`	`[1]`	`[1]`
`[1, 1]`	1	0	1	0
`[1]`	1	0	0	0
`[1, 1]`	0	1	0	1
`[1]`	1	0	0	0

Diese Matrix stellt die Zahl 6677 (binär 1101000010101) dar und wird durch `[[1, 1], [1], [1, 1], [1]]` und `[[2, 1], [1], [1], [1]]` codiert.

Schreiben Sie eine Funktion `decode`, die solche Spezifikationen als Argument bekommt und die die so codierte Zahl zurückgibt. Der Aufruf

```
decode([[1, 1], [1], [1, 1], [1]], [[2, 1], [1], [1], [1]])
```

soll also z.B. das Ergebnis `6677` liefern. Zu der Aufgabe gehört eine Datei mit diversen Ein- und Ausgaben zum Testen.

Anmerkungen:

(i) Spezifikationen sind nicht immer eindeutig. Z.B. würden `[[2], [1], [0]]` und `[[1], [1], [1]]` die Matrix beschreiben, die die Zahl 98 codiert. Es gibt aber noch sieben andere 3×3-Matrizen, die durch dieselbe Spezifikation beschrieben werden. Sie können für Ihre Funktion `decode` jedoch davon ausgehen, dass der gegnerische Geheimdienst nur eindeutige Spezifikationen benutzt.

(ii) Die Aufgabe gilt als gelöst, wenn Ihre Funktion Matrizen bis zur Größe 4×4 verarbeiten kann. Es gibt 65536 verschiedene binäre Matrizen dieser Größe; man kann also so arbeiten, dass man einfach alle möglichen Lösungen durchprobiert.

(iii) Schreiben Sie *optional* Ihre Funktion so, dass sie auch mit größeren Matrizen* klarkommt. Dafür bietet sich eine Technik wie *Backtracking* an.

*Auch dafür finden Sie in der og. Datei Beispiele.

Zur Abschätzung des Arbeitsaufwandes hier die Nettozeilenzahlen (ohne Leerzeilen, Kommentare und `import`-Befehle) meiner Musterlösungen:

- Aufgabe 1: 23 Zeilen
- Aufgabe 2: 47 Zeilen
- Aufgabe 3: 65 Zeilen
- Aufgabe 4: 79 Zeilen
- Aufgabe 5: 26 Zeilen (einfache Lösung) bzw. 71 Zeilen (Backtracking)