EFME 2010 LU Exercise 3

Report

Gruppe 15:

Fritz Daniel - 0507049 - 935

Hiller Elias - 0525787 - 935

Sonderegger Josef - 0501625 - 935

Bayes Theorem

Um das Bayes Theorem anwenden zu können, ist die Wahrscheinlichkeitsverteilung und die Randdichten gegeben.
Als Betrachtungsbereich wurde -5 bis 10 gewählt, jeweils im 0.01 Schritte Intervall, da N(1.5,1) und N(3,1) Großteils in diesem Bereich liegen.

Berechnen der Bedingten Wahrscheinlichkeit $p(x|\omega)$

Funktion : *calc_pxw(pw,scale,mean,sigma)*
Mit der Funktion *normpdf* wird mit dem gewählten Betrachtungsbereich -5:0.01:10 und den jeweils gegebenem Mittelwert und Standardabweichung die Normaldichte berechnet und schließlich mit der ebenfalls gegebenen Wahrscheinlichkeit multipliziert.

Berechnen der Marginalverteilung $p(x)$

Funktion : *calc_marginaldistribution(pxw1,pxw2)*
Die Randverteilung lässt sich einfach mittels Aufsummieren der bedingten Wahrscheinlichkeiten $p(x \mid \omega_1)$ und $p(x \mid \omega_2)$ errechnen.

Berechnen der A-Posteriori-Wahrscheinlichkeit

Funktion : *calc_posterior(pxw,px)*
Für das Berechnen der A-Posteriori-Wahrscheinlichkeit ist die Bedingte Wahrscheinlichkeit der Wahrscheinlichkeit ω_n und die Randverteilung von Nöten und erfolgt durch zeilenweise Division von Ersterer durch Zweitere.

Berechnen der Fehlerwahrscheinlichkeit an einer bestimmten Grenze x_p

Funktion : *calc_errorrate(pw1,boundary)*
In unserem Beispiel war diese Grenze an der Stelle x_p= 4. Die Fehlerrate wird dabei mittels folgender Formel berechnet:

$$P(error) = \int P(error|x)^* p(x)\,dx$$

wobei

$P(error|x) =$

- $P(x|\omega_2)$, falls $\omega 1$
- $P(x|\omega_1)$, falls $\omega 2$

Die dabei berechnete Fehlerwahrscheinlichkeit beträgt :

- $P(\omega_1)=0.5: P(error|x) = 0.4245$ => 42,5%
- $P(\omega_1)=0.9: P(error|x) = 0.0910$ => 9,1%

Berechnen der Bayes Fehlerwahrscheinlichkeit:
Funktion : calc_bayeserrorrate(pw1)
Die Bayes Errorrate wird mittels folgender Formel berechnet:

$$P(error|x) = 1 - max_j P(\omega_j|x)$$

Da aber nur zwei Wahrscheinlichkeiten gegeben waren, haben wir jeweils einfach das Minimum als Wert herangezogen.

Die Fehlerwahrscheinlichkeiten betragen:

$P(\omega_1) = 0.9: P(error|x) = 0.0895$ => 22,6%
$P(\omega_1) = 0.5: P(error|x) = 0.2263$ => 9,0%

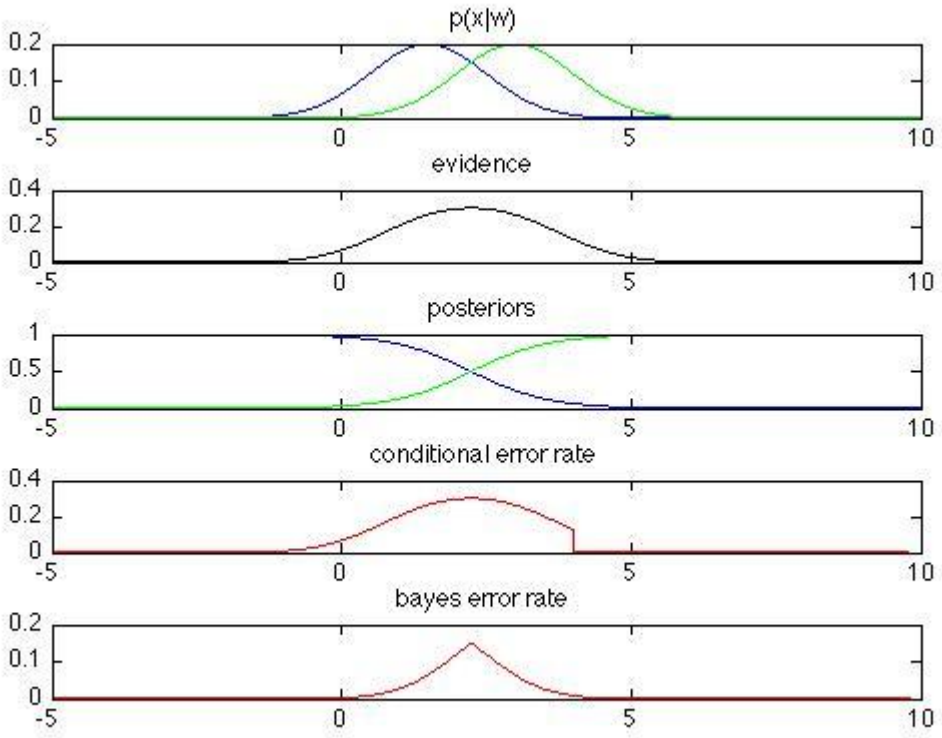

Abbildung 1.1: $P(\omega_1) = 0.5, P(\omega_2) = 0.5$

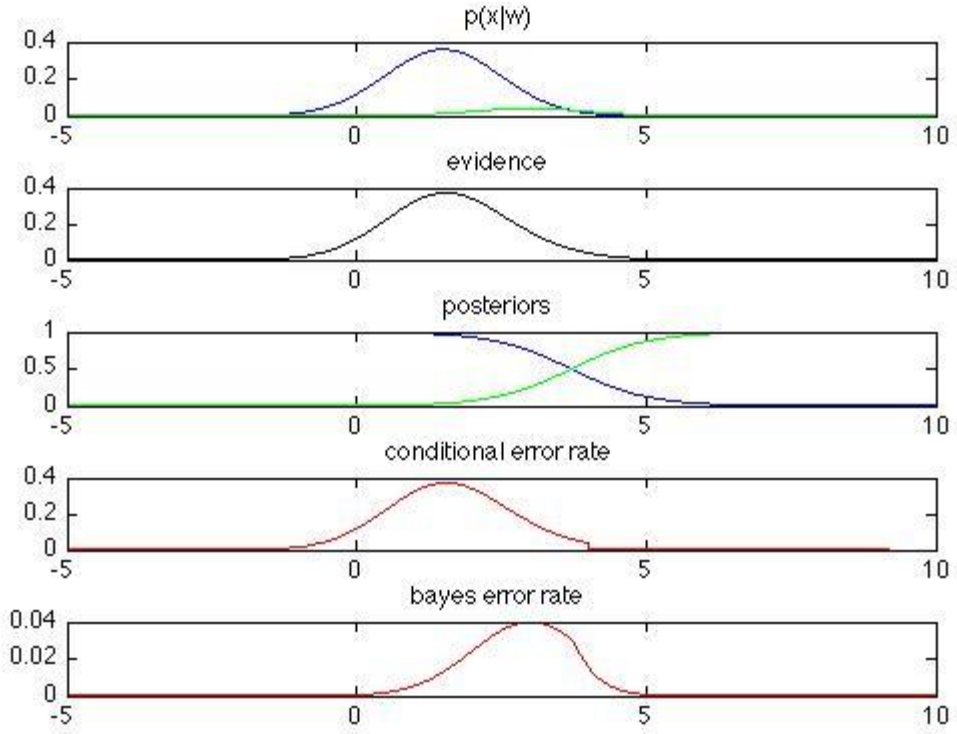

Abbildung 1.2: $P(\omega_1) = 0.9, P(\omega_2) = 0.1$

Die geringeren Fehlerwahrscheinlichkeit bei einer Wahrscheinlichkeitsverteilung von $P(\omega_1) = 0.9$ kann damit erklärt werden, dass die zweite Klasse eine deutlich geringere Gewichtung erhält.

Discriminant Functions for the Normal Density

Gegeben waren zwei verschiedene Sets

A = [2 2 4 4; 6 0 6 0];

B = [8 8 10 10; 2 4 2 4];

Zuerst wurde die Diskriminantenfunktion von Hand mittels der Kovarianzmatrix und zum Vergleich auch mit der Einheitsmatrix berechnet.
Die Kovarianzmatrix berechnet sich für jede Spalte eines Sets mit der Formel

$$C = \frac{1}{N-1} \sum_{i=1}^{N} (x_i - m)(x_i - m)^T$$

Das m ist hierbei einfach der Mittelwert des jeweiligen Sets. Die Diskriminantenfunktion wird aus der Formel $g_j(x) = -\frac{1}{2} d_j^2(x) + \left[-\frac{1}{2} ln \left| \sum_j \right| + ln\, P(\omega_j) \right]$ berechnet.

Die Mahalanobisdistanz wird wie folgt berechnet.

$$d_j^2(x) = (x - \mu_j)^T \sum_j^{-1} (x - \mu_j)$$

Rechengang, sowie Ergebnisse sind als Fotos in der Abgabe beigelegt.

Im Anschluss daran wurden diese Ergebnisse via Matlab ausgegeben. Als erstes sollten die beiden Sets und die Mittelwerte mit `gscatter` eingezeichnet werden. Nach Wunsch soll die Ausgabe angepasst werden, damit man die Ergebnisse gut ersichtlich vergleichen kann. Die errechneten Funktionen können einfach mit `ezplot` ausgegeben werden.

Die beiden Sets wurden in zwei Gruppen (1 und 2) unterteilt und ausgegeben. Egal ob nun die Berechnung mittels Kovarianzmatrix oder Einheitsmatrix durchgeführt wird, die Diskriminantenfunktion teilt immer die beiden Sets. Bei Ersterem mit einer quadratischen Funktion, bei Letzterem ist die Funktion linear.

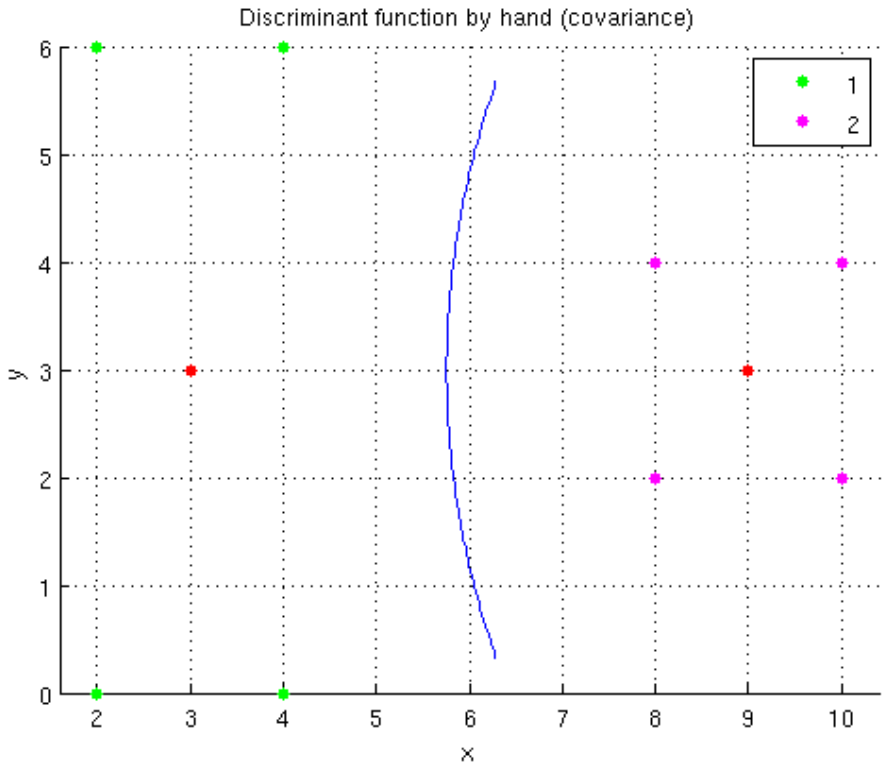

Abbildung 2.1: Sets, Mittelwerte und Diskriminantenfunktion - Berechnung mit Kovarianzmatrix von Hand.

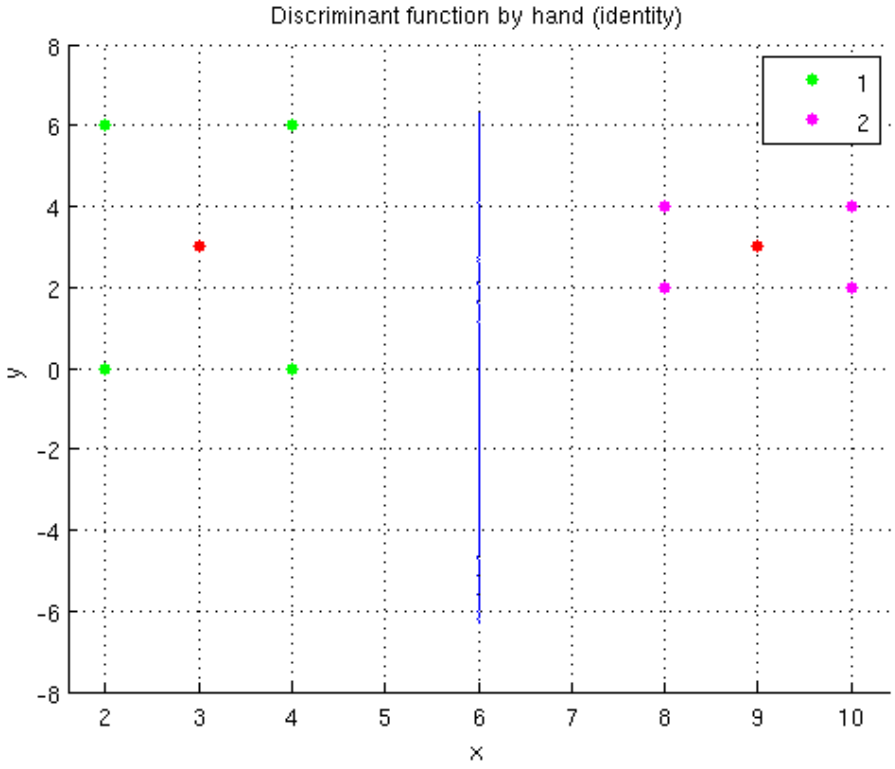

Abbildung 2.2: Sets, Mittelwerte und Diskriminantenfunktion - Berechnung mit Einheitsmatrix von Hand.

Um die Diskriminantenfunktion zu berechnen werden die Koeffizienten aus der `classify` Methode verwendet.

```
[X,Y] = meshgrid(linspace(1,11),linspace(-1,7));
X = X(:); Y = Y(:);
[c,err,P,logp,coeff] = classify([X Y],C',gr,'quadratic');
gscatter(X,Y,c);

% Draw boundary between two regions - using testsets
K = coeff(1,2).const;
L = coeff(1,2).linear;
Q = coeff(1,2).quadratic;
f = sprintf('0 = %g + %g*x + %g*y + %g*x^2 + %g*x.*y +
%g*y.^2', K,L,Q(1,1),Q(1,2)+Q(2,1),Q(2,2));
```

Dabei wird eine quadratische Gleichung aus diesen Koeffizienten aufgebaut und wie schon bei den händischen Beispielen mit `ezplot` ausgegeben. Werden die beiden Mittelwerte verbunden, kann man erkennen, dass die Diskriminantenfunktion genau an dessen Scheitelpunkt geschnitten wird. Um die Ergebnisse der Einheitsmatrix zu erhalten wird im `classify` einfach `linear` verwendet.

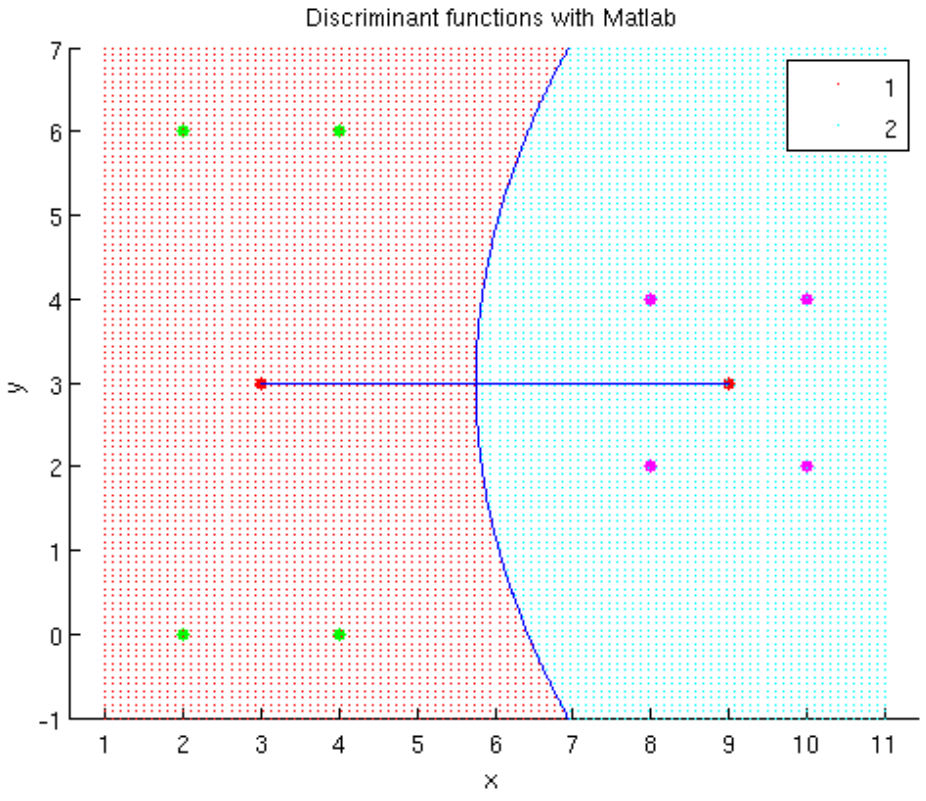

Abbildung 2.3: Sets, Mittelwerte und Diskriminantenfunktion - Berechnung mit Kovarianzmatrix mit Matlab

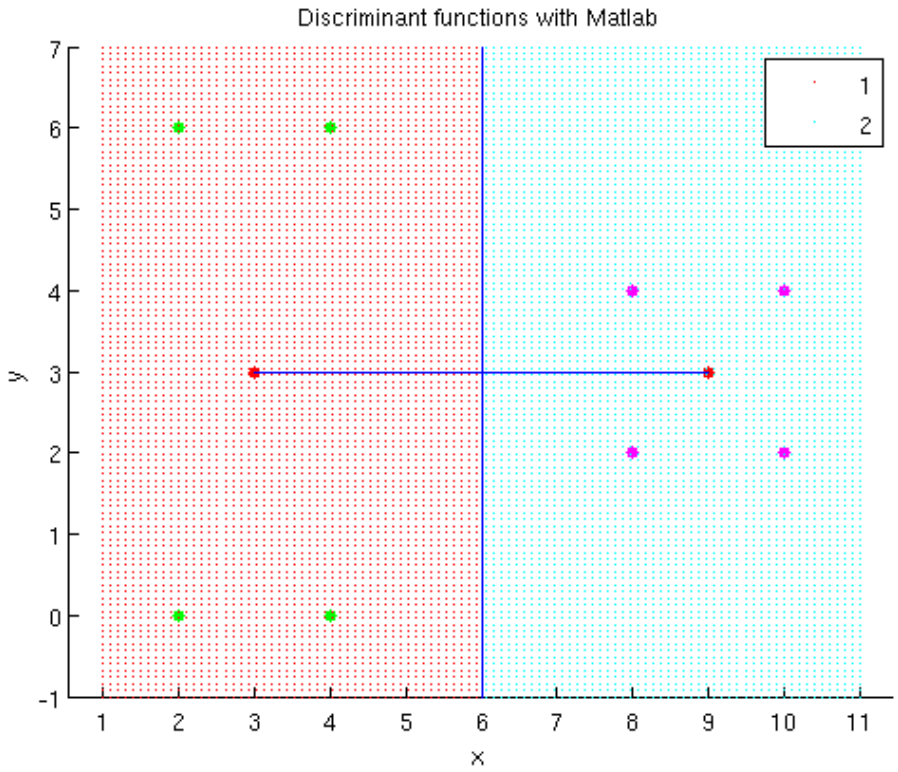

Abbildung 2.4: Sets, Mittelwerte und Diskriminantenfunktion - Berechnung mit Einheitsmatrix in Matlab

Vergleicht man nun die Ergebnisse, so erkennt man, dass diese ident sind. Daraus schließen wir, dass die handschriftlichen Ergebnisse korrekt sind. Bei größeren Sets ist dies sicher nicht mehr so einfach zu berechnen und vor allem zu kontrollieren. Weiters wird zur Berechnung der Kovarianzmatrix nur der Schätzer verwendet, was bei nicht so eindeutigen Testsets wahrscheinlich schlechtere Ergebnisse liefert.

Im zweiten Teil der Aufgabe sollen zwei Multivariate Normalverteilungsfunktionen mittels μ = [5 6 5] und μ = [0 1 1] erstellt werden. Dafür wird bei der Funktion `mvnrnd` die Einheitsmatrix verwendet um die Kovarianzmatrix zu berechnen. Wir haben hier 20 Punkte für jedes Set erstellt. Diese Punkte werden wieder in zwei Gruppen unterteilt und in diesem Fall mit `scatter3` in der dreidimensionalen Ebene ausgegeben. Vom berechneten Mittelwert für beide Gruppen wird eine Linie gezogen.

```
[C, err, P, logp, coeff] = classify(u, u, gr, 'quadratic');
K = coeff(1,2).const;
L = coeff(1,2).linear;
Q = coeff(1,2).quadratic

f = @(x,y) (K + L(1)*x + L(2)*y)/-L(3);
ezmesh(f);
```

Hier wird die Diskriminantenfunktion im Raum berechnet und dann ausgegeben. Die Funktion teilt die beiden Sets genau in der Mitte, wie man an der Linie, der Verbindung der Mittelwerte, schön erkennen kann.

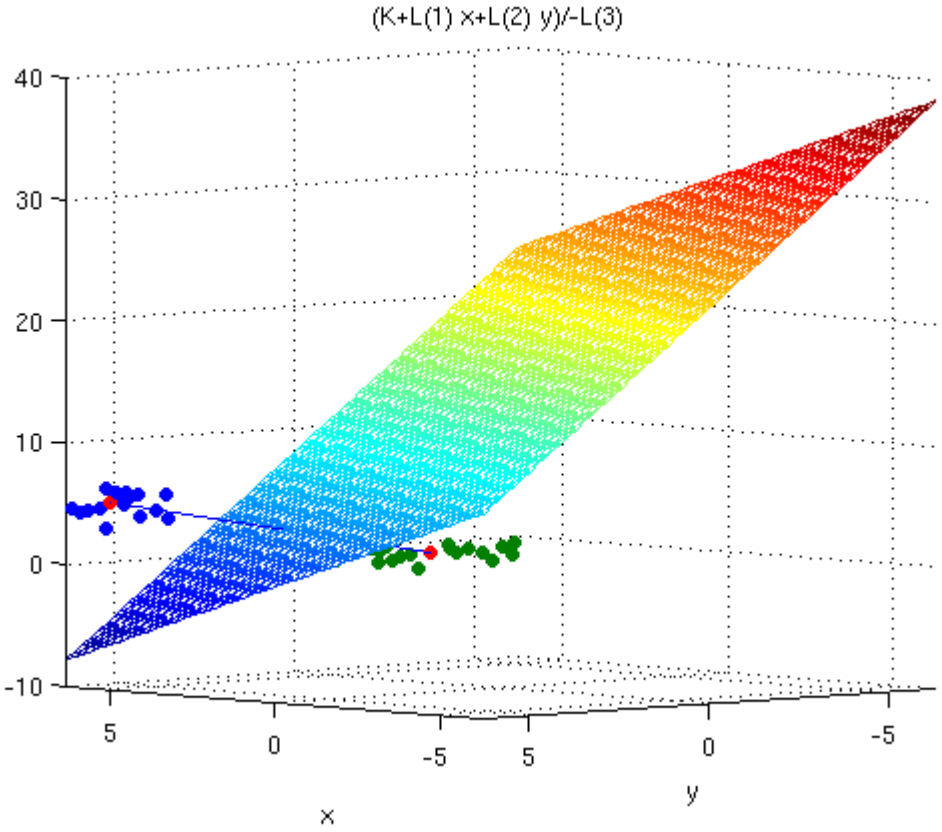

Abbildung 2.5: Diskriminantenfunktion 3d

Eigenfaces

Die Bilder wurden händisch normalisiert, ausgeschnitten (115 x 130 Pixel = img_x x img_y) und schließlich als GIFs abgespeichert. Dabei wurde darauf geachtet, dass bei allen Bilder die Augen bzw. Nasen auf derselben Höhe sind (siehe Abbildung 3.1). Da für die Rekonstruktion nur das Gesicht nötig und wichtig ist wurden Haare usw. weggeschnitten. Zu Problemen kam es zuerst da die Bilder als PNGs bzw. danach als GIFs abgespeichert wurden, weil Photoshop bei den Standardeinstellungen einen Farbreduktionsalgorithmus verwendet. Dies ist mit freiem Auge im Bildbetrachter nicht erkennbar, sehr wohl aber wenn das Bild mit Matlab geöffnet wird. Die Umstellung auf Graustufen beim Exportieren im Photoshop brachte das gewünschte Ergebnis.

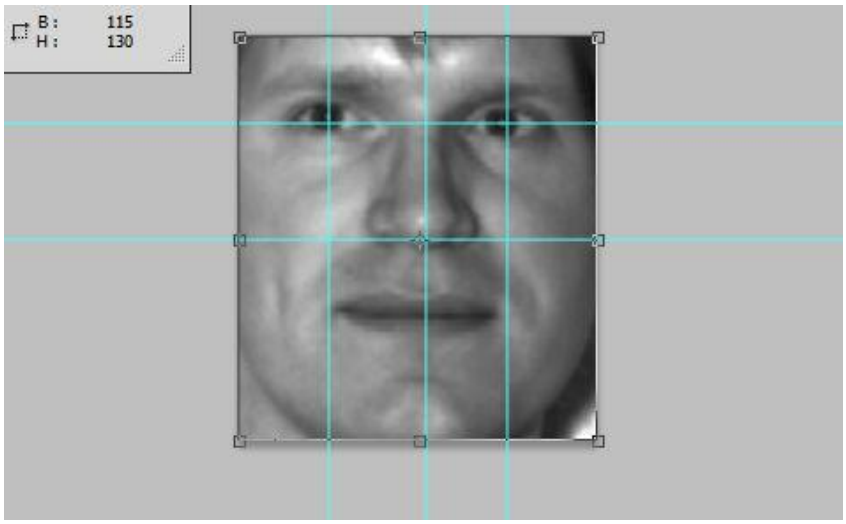

Abbildung 3.1: Normalisierung der Bilder in Photoshop

Es wurden verschiedene Test und Trainingssets verwendet und ausprobiert. Zum Beispiel das Bild *„Subject01_normal.gif“* als Testbild und alle anderen als Trainingsbilder (*abgabe3_bsp3.m*). Andere Kombinationen waren alle Bilder von *„Subject01“ (abgabe3_bsp3_training_subject01.m)*, alle Bilder mit Brille *„glasses“* oder alle Bilder mit *„light“* im Namen als Testset zu verwenden und dementsprechend die restlichen Bilder als Trainingsset. Für die weiteren Bilder hier wurde das Testset: *„Subject01_normal.gif“* verwendet, wenn nicht anders angegeben.

Berechnung der Eigenfaces - Ablauf

Die Bilder werden geladen und mittels `reshape` in eine Spalte gepresst. Zum Betrachten der Bilder wurde die Funktion `viewcolumn` geschrieben. Schließlich wurde das Meanface berechnet. Als nächstes wird von allen Trainingsbildern das Meanface subtrahiert und in der Matrix *A* gespeichert. Zur Berechnung der Eigenvalues und Vectors wurde der Transpose Trick angewendet. Dieser hat den Vorteil, dass die Kovarianzmatrix nur mehr die Dimension Anzahl der Bilder und nicht mehr Anzahl der Pixel hat (also bei uns z.B. 69x69 statt 14950x14950) und dadurch die Berechnung viel schneller gemacht werden kann bzw. der Speicher ausreichte (kein

„out of memory“). Es werden also die Eigenvalues und Vectors von `cov = A' * A` mittels `eig(cov)` berechnet und mit der function `eigsort` sortiert. Und zwar werden dort zuerst die Diagonalwerte von Eigenvalues absteigend sortiert und dementsprechend die dazugehörigen Eigenvectors. Um die benötigten Vektoren für `A * A'` zu bekommen werden die sortierten Vektoren mit der Matrix `A` multipliziert, in `U` gespeichert und anschließen mit `normc(U)` normalisiert. Die Eigenfaces werden in Abbildung 3.4 dargestellt.

Diskussion:

1. Entsprechen das *mean face* und die *eigenfaces* dem, was erwartet wurde?

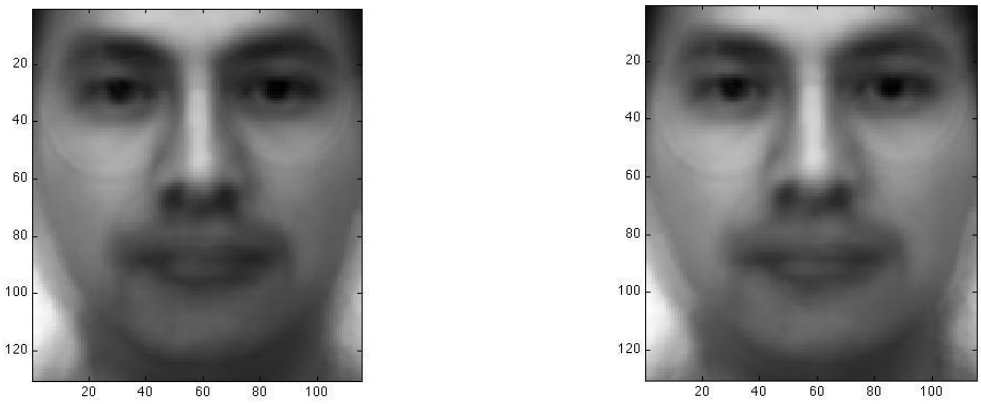

Abbildung 3.2: *links*: mean face - *rechts*: mean face inklusive den eigenen Bildern

Das Durchschnittsgesicht sieht dem einer echten Person sehr ähnlich (Abbildung 3.2) und entspricht den Erwartungen. Lediglich aufgrund der verschwommenen Erscheinung und den leicht sichtbaren Rändern der Brille erscheint dieses etwas unnatürlich. Ein Bild aus *A* mit dem Meanface addiert ergibt wieder das ursprüngliche Bild, wie in Abbildung 3.3 ersichtlich.

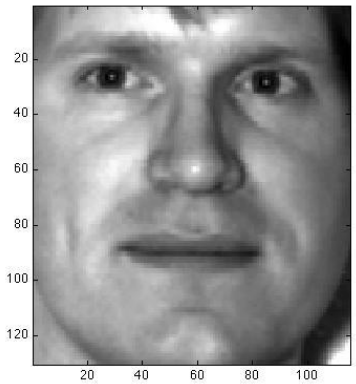

Abbildung 3.3: Mean-subtrahiertes Trainingsbild + Meanface

Die Eigenfaces (Abbildung 3.4) hingegen haben keine wirkliche Ähnlichkeit mit dem echten Gesicht. Dies soll natürlich auch nicht Sinn der Sache sein. Wenngleich einige dieser Gesichter ein schlechtes Kontrastverhältnis aufweisen, entsprechen sie durchaus den erwarteten Werten.

Auffallend war, dass am Ende der sortierten Eigenfaces sich ein paar extreme Ausreißer befinden die das Ergebnis der Rekonstruktion noch negativ beeinflussen können.

Zum Beispiel die letzten 5 Werte der sortierten Eigenvalues:

430218,097
409480,705
2,316 e-09
8,100 e-10
-6,269 e-09

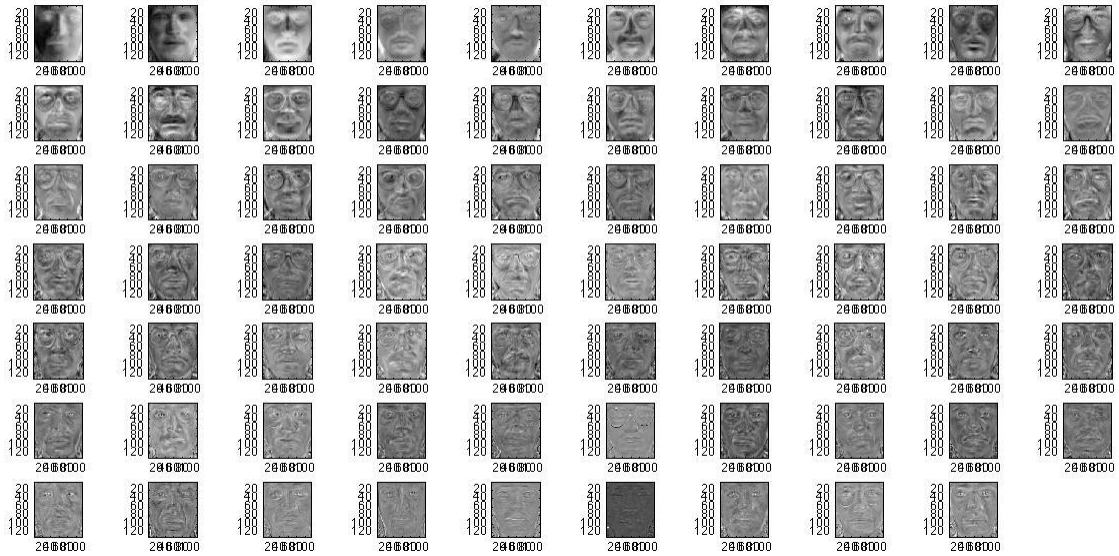

Abbildung 3.4 : Alle Eigenfaces

2. Schreiben sie eine Funktion, welche die Ausdehnungskoeffizienten eines Bildes im Bezug auf Eigenfaces berechnet. Als Input soll die Eigenfacematrix und das Bild dienen. Rückgabewert soll ein Vektor mit den Hauptbauteilen sein.

```
function[comp] = coeff(U, test, mean_img)
    comp = U' * (test - mean_img);
end
```

Hier wurde auch das Meanface übergeben, da es für die Berechnung benötigt wird. Die Berechnung ist sehr einfach. Die Matrix wird transponiert und schließlich mit dem Testbild ohne das Meanface multipliziert.

3. Schreiben sie eine Funktion, welche ein Gesicht mittels einer limitierten Anzahl an Komponenten rekonstruieren kann und dieses schließlich zurückgibt.

```
function[recface] = reconstruction(U,comp, mean_img, usedcomp)
    recface = U(:, 1:usedcomp) * comp(1:usedcomp) + mean_img;
end
```

Abbildung 3.5 : *links*: Originalbild - *rechts*: Rekonstruiertes Gesicht mit 10 Komponenten
unten: Rekonstruiertes Gesicht mit 65 Komponenten (alle waren im Trainingsset)

Mit `usedcomp` wird eine Zahl an die Funktion übergeben, die angibt, wie viele Komponenten für die Rekonstruktion angewendet werden sollen. Wichtig ist auch, dass hier das Meanface addiert werden muss, da es von den Bildern in der Matrix abgezogen wurde.
Wie in Abbildung 3.5 ersichtlich wird funktioniert die Rekonstruktion mit 10 Komponenten zwar schon recht ordentlich, mit 65 Komponenten kann jedoch kaum mehr ein Unterschied zum Originalbild wahrgenommen werden.

4. Schreiben sie eine Prozedur, welche die Rekonstruktion des Gesichtes animiert.

```
for i=1 : usedcomp
        viewcolumn(reconstruction(U,comp, mean_img, i));
        pause(0.1)
        disp([num2str(i), ' von ', num2str(usedcomp)])
end
```

Bei dieser Funktion wird das gleiche Bild immer wieder rekonstruiert, jedoch immer mit einer Komponente mehr als zuvor. Damit dies nicht zu schnell geschieht, wird mittels `pause(0.1)` eine Pause zwischen den Bildern eingelegt.
Mit steigender Anzahl an Features wird auch das rekonstruierte Bild immer besser. Allerdings scheinen zumindest die letzten 3 Bilder solche Ausreißer zu sein, dass das Ergebnis schlussendlich deutlich verschlechtert wird.

5. Laden sie das Testgesicht und finden sie heraus, wie gut die Eigenfaces genügen um ein neues Gesicht zu rekonstruieren, welches nicht im originalen Datenset enthalten war.

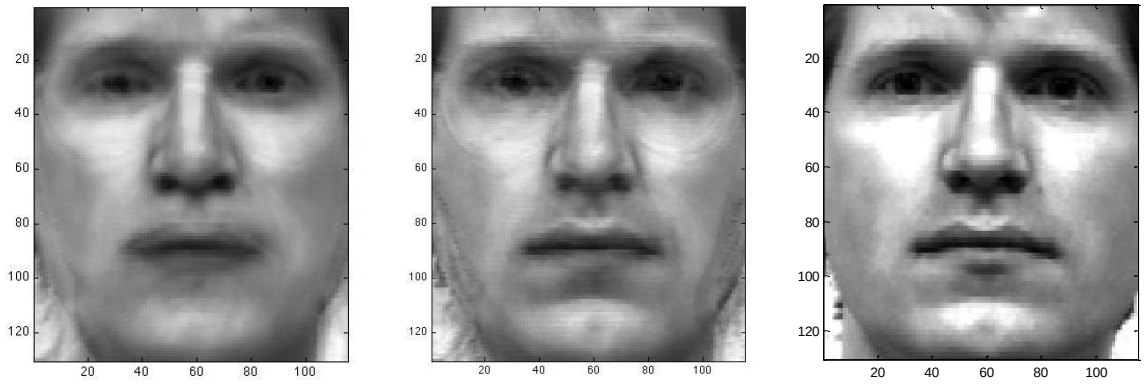

Abbildung 3.5 : Trainingsset mit 9 anderen Bildern von „Subject01“ (ohne Testbild)
links: Rekonstruktion mit 10 Komponenten
Mitte: Rekonstruktion mit 65 Komponenten *rechts:* Original

Auch die Rekonstruktion eines Gesichtes, welches so nicht im Trainingsset enthalten war funktioniert recht gut. Werden weniger Komponenten verwendet, so ist das Bild wesentlich unschärfer. Benutzt man mehr Komponenten so werden einige Details schärfer, jedoch machen sich Fehler, wie etwa die Brillenränder deutlicher bemerkbar. Die Ähnlichkeit mit dem Original ist klar sichtbar bei den Augen, Nase und Lippen. Die gute Rekonstruktion ist vor Allem darauf zurück zu führen, dass zwar dieses Bild *„Subject01_normal.gif“* nicht im Trainingsset war aber 9 andere Bilder dieser Person. Im Vergleich dazu wenn wieder das Bild *„Subject01_normal.gif“* rekonstruiert werden soll, das Trainingsset aber gar kein Bild von Subject01 enthält bekommt man das Ergebnis in Abbildung 3.6. Hier kann nicht mehr von einer Ähnlichkeit gesprochen werden, diese Rekonstruktion ist unbrauchbar.

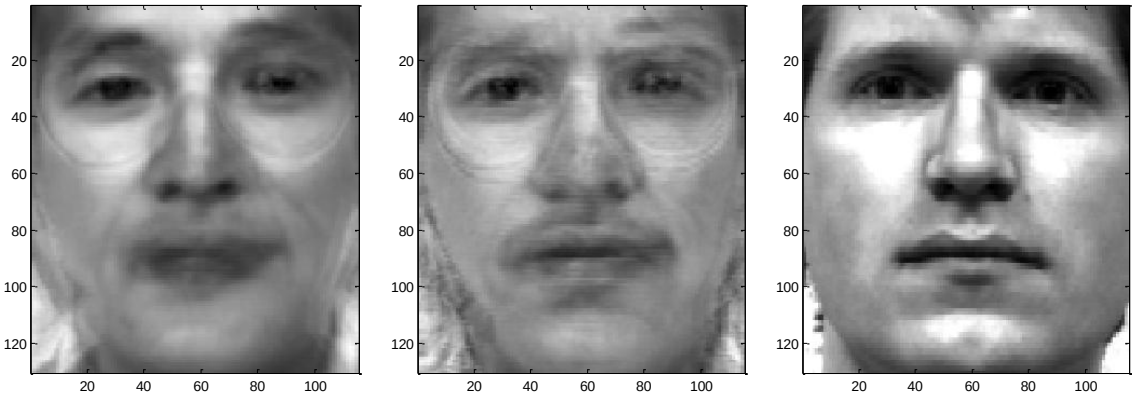

Abbildung 3.6 : Trainingsset ohne „Subject01“ - *links*: Rekonstruktion mit 10 Komponenten *Mitte*: Rekonstruktion mit 60 Komponenten *rechts:* Original

6. Fügen Sie ihre eigenen Gesichter zur Datenbank hinzu und zeigen sie die Rekonstruktion anhand ihres eigenen Gesichtes.

Um das eigene Gesicht (*„Subject11“*) zur Datenbank hinzuzufügen wurden 3 unterschiedliche Aufnahmen gemacht. Die Belichtungsverhältnisse wurden mittels Photoshop künstlich erzeugt.(Abbildung 3.7)

Eine Abbildung aller Eigenfaces, inklusive den eigenen, ist in Abbildung 3.8 ersichtlich.

Während die Rekonstruktion des eigenen Bildes schon bei 10 Komponenten vielversprechend aussieht, ist sie mit 70 Komponenten noch schärfer und beherbergt mehr Details (Abbildung 3.9). Die Bilder, inklusive Testbild, befanden sich im Trainingsset und darum die sehr gute Rekonstruktion. Wird wiederum versucht das eigene Bild zu rekonstruieren, wenn es sich nicht im Trainingsset befindet, kommt es zu den Ergebnissen in Abbildung 3.10. Die Rekonstruktion ist auch in diesem Fall unbrauchbar aus den oben genannten Gründen.

Abbildung 3.7: Die eigenen Bilder in allen Variationen

Abbildung 3.8: Eigenfaces inklusive den eigenen Bildern

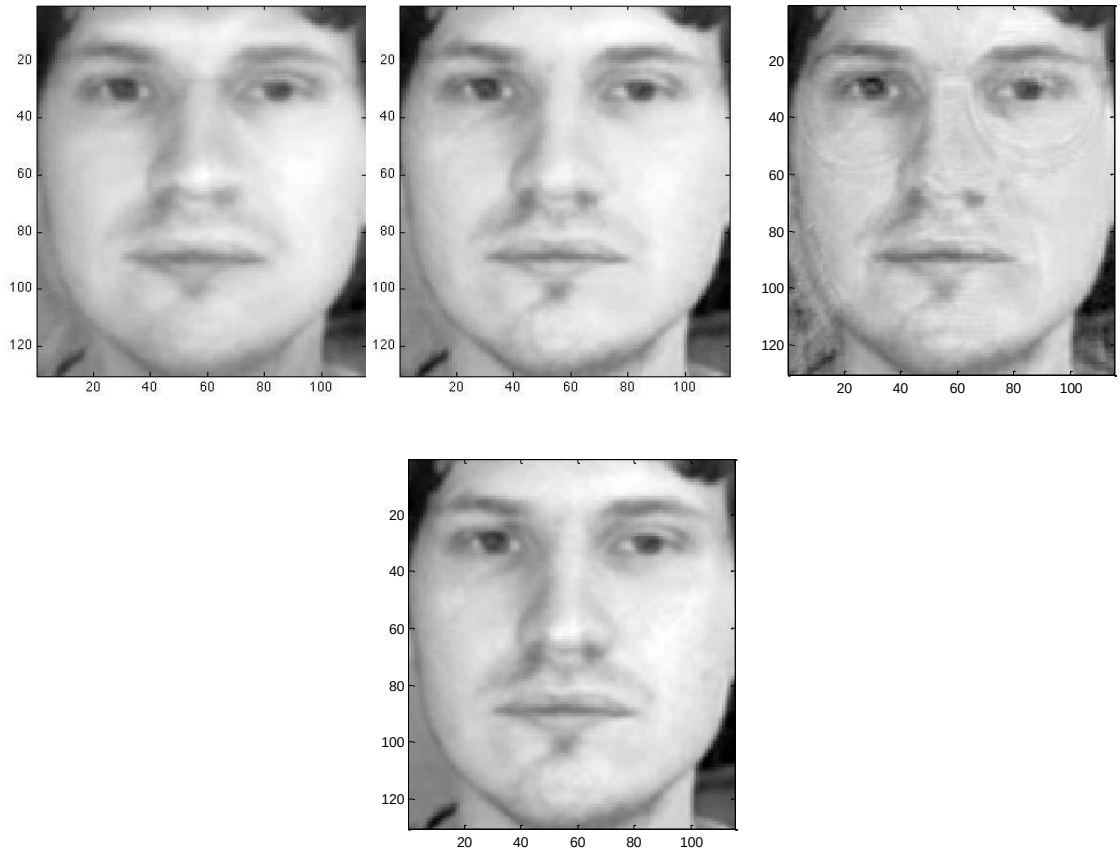

Abbildung 3.9: *links:* rekonstruiert mit 10 Komponenten *Mitte:* rekonstruiert mit 70 Komponenten *rechts:* rekonstruiert mit 76 Komponenten *unten:* Original

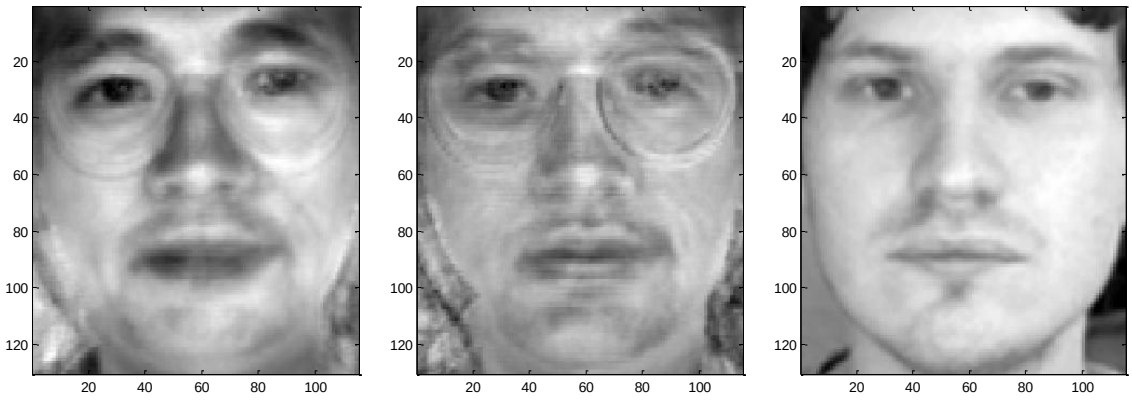

Abbildung 3.10: Trainingsset ohne *„Subject11“* - *links:* rekonstruiert mit 10 Komponenten *Mitte:* rekonstruiert mit 50 Komponenten *rechts:* Original