Dimensionsreduktion

Joern Ploennigs · AI4SC

> To tell you the truth, I've never met anybody who can envision more than three dimensions.
>
> — Brian Greene

Methoden zur *Dimensionsreduktion* werden im Machine Learning immer dann genutzt, wenn Datensätze vereinfacht werden sollen, um sie besser zu verstehen oder schneller zu erlernen. Hierbei teilt man die Verfahren in zwei Anwendungsfälle:

- Reduktion der Anzahl an Variablen in sehr großen Datensätzen mittels der **Hauptkomponentenanalyse**
- Reduktion der Anzahl an Beobachtungen, um typische Muster abzuleiten mittels **Matrixfaktorisierung**

Dimensionsreduktion wird häufig in der Vorverarbeitung von ML-Modellen eingesetzt, um etwa *Overfitting* zu vermeiden, die Visualisierung zu erleichtern oder Trainingszeiten zu verkürzen. So können die Verfahren zur Reduktion von Sensor-Messwerten in Brücken oder Gebäuden benutzt werden oder zur Verdichtung komplexer Materialprüfdaten auf wenige Hauptkomponenten.

Zum Vergleich: PCA sucht lineare Richtungen maximaler Varianz und eignet sich besonders zur Verdichtung stark korrelierter Variablen. NMF zerlegt einen Datensatz dagegen in additive, nichtnegative Bausteine und liefert dadurch oft besser interpretierbare Muster oder Prototypen. PCA ist also meist das Mittel der Wahl für kompakte Projektionen, NMF eher für interpretierbare Komponenten in nichtnegativen Daten.

Hauptkomponentenanalyse (PCA)

Methode

Die Hauptkomponentenanalyse (en. *Principal Component Analysis*, PCA) wird eingesetzt, um Datensätze mit vielen Variablen zu vereinfachen, damit wir sie besser interpretieren und nutzen

können und damit sich kompaktere Modelle trainieren lassen. Das ist insbesondere bei Datensätzen mit vielen, stark korrelierten Variablen sinnvoll, da sowohl Nutzer die vielen Variablen nur schwer verstehen als auch viele lineare Modelle bei stark korrelierten Variablen instabil werden oder zusätzliche Regularisierung benötigen.

Stellen wir uns vor, wir überwachen eine Brücke mit 10 verschiedenen Sensoren. Viele der Messreihen beschreiben ähnliche Effekte und sind stark korreliert. PCA hilft dabei, die wichtigsten Muster, z.B. eine Vibration in X/Y-Zugrichtung, aus diesen Daten herauszufiltern und auf 2 bis 3 Hauptachsen zu reduzieren.

Karl Pearson beschrieb die Methode bereits 1901 [1]. Ziel ist es, eine Matrix mit vielen Spalten (Dimensionen) auf einige wenige lineare Komponenten (Hauptkomponenten) zu reduzieren, die den Datensatz möglichst gut beschreiben. Dafür suchen wir neue, synthetische Achsen, entlang derer die Varianz im Datensatz möglichst hoch ist. Gleichzeitig werden die Daten dekorreliert. Anders als bei linearen Regressionsmodellen ist es hierbei nicht das Ziel, eine einzelne Ausgangsvariable zu modellieren, sondern den vollständigen Datensatz.

Wir betrachten den bekannten Energie- und Wetter Datensatz.

```python
import numpy as np # Import von NumPy
import pandas as pd # Import von Pandas

egywth = pd.read_csv("../data/UROS/Energy1D_weather_clean.csv",
parse_dates=[0])
egywth.head()
```

	EVDateE_N4E_7AE_ISV_IDASILAMEOSS_DATUM	QN_3	…	TMK	UPM	TXK	TNK	TGK	eor	Temperatur-Klasse	Heiz-Kuehl-Tage	QN_4	NF_4
0	2020-12-30 00:00:00+00:00 NaN NaN NaN NaN NaN 2020-12-30 00:00:00+00:00 4177 20201230 10.0		…	4.0	81.0	4.6	2.9	2.2	eor	Cold	Heizgradtag	nicht alle Parameter korrigiert	5
1	2020-12-31 00:00:00+00:00 NaN NaN 1256291.05.0 2020-12-31 00:00:00+00:00 4177 20201231 10.0		…	3.4	83.0	4.4	2.3	0.9	eor	Cold	Heizg-	nicht al-	5

	EV_BE_NE_TAB_LSV_DESLIAMEOSIS_DATUM	QN_3	...TMK	UPM	TXK	TNK	TGK	eor	Temperatur-Klasse	Heiz-Kuehl-Tage	QNS_4	QNF_4
										radtag	le Parameter korrigiert	
2	202100:1-0108012212990.0 1.0 00:00:00+00:00 2021421-202100101...0 00:00:00+00:00	2.0	90.0	3.0	1.1	0.3	eor	Cold	Heizgradtag	nicht alle Parameter korrigiert	5	
3	2021-11-0263012432384.0 2.0 00:00:00+00:00 2021421-202100102...0 00:00:00+00:00	3.3	95.0	4.0	2.6	2.0	eor	Cold	Heizgradtag	nicht alle Parameter korrigiert	5	
4	202100:1-0375012222383.0 2.0 00:00:00+00:00 2021421-202100103...0 00:00:00+00:00	3.5	81.0	4.6	2.4	0.8	eor	Cold	Heizgradtag	nicht alle Parame-	5	

EV_HT_740	EV_NT_740	E_AV_Lab	E_SV_Lab	ES_Lab	FX	FM	RSK	RSKF	SDK	SHK_TAG	NM	VPM	PM	TMK	UPM	TXK	TNK	TGK	eor	Temperatur-Klasse	Heiz-Kühl-Tage	QNS	QNF_4
																							ter kor-ri-giert

Wir analysieren vom Datensatz allerdings nur die numerischen, nicht-NA Spalten, weshalb wir sie selektieren.

```
egywthNumber = egywth.select_dtypes(include='number').dropna()
egywthNumber = egywthNumber[['EV_HT_740', 'EV_NT_740', 'E_AV_Lab', 'E_SV_Lab',
'ES_Lab', 'FX', 'FM', 'RSK', 'RSKF', 'SDK', 'SHK_TAG', 'NM', 'VPM', 'PM',
'TMK', 'UPM', 'TXK', 'TNK', 'TGK']]
```

Schritt 1 - Datenzentrierung: Die Daten werden zentriert, indem der Mittelwert jeder Dimension subtrahiert wird. Dadurch liegen alle Daten um den Ursprung herum.

$$\boldsymbol{X}_{\text{cent}} = \boldsymbol{X} - \boldsymbol{\mu}_X.$$

```
egywthCentered = egywthNumber - egywthNumber.mean()
egywthCentered.head()
```

	EV_HT_740	EV_NT_740	E_AV_Lab	E_SV_Lab	ES_Lab	FX	FM	RSK	RSKF	SDK	SHK_TAG	NM	VPM	PM	TMK	UPM	TXK	TNK	TGK
2	−1245																		
3	−75.8																		
4	−1249																		
5	1994.																		
6	2224.																		

Schritt 2 - Berechnung der Kovarianzmatrix: Die Kovarianzmatrix C wird berechnet, um die Streuung und Korrelation zwischen den verschiedenen Dimensionen zu erfassen.

$$\boldsymbol{C} = \frac{\boldsymbol{X}_{\text{cent}}^T \boldsymbol{X}_{\text{cent}}}{n-1}.$$

```
cov_matrix = np.cov(egywthCentered, rowvar=False)
```

Schritt 3 - Eigenwertzerlegung der Kovarianzmatrix: Die Kovarianzmatrix wird einer Eigenwertzerlegung unterzogen, um die Eigenwerte und Eigenvektoren zu berechnen. Die Eigenvektoren repräsentieren die Richtungen der größten Varianz, und die Eigenwerte repräsentieren die Größe der Varianz in diesen Richtungen. Der Eigenvektor einer quadratischen Matrix ist ein Vektor, der durch die Matrix nur um einen skalaren Faktor, den Eigenwert λ, gestreckt oder gestaucht wird. Mathematisch ausgedrückt:

$$\boldsymbol{C}\boldsymbol{v} = \lambda \boldsymbol{v}$$

dabei ist λ der skalare Eigenwert und $\boldsymbol{v}$ der zugehörige Eigenvektor.

```
# Berechne die Eigenwerte und Eigenvektoren
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
```

Schritt 4 - Auswahl der Hauptkomponenten: Die Hauptkomponenten (*Principal Components*) sind die Eigenvektoren, die den größten Eigenwerten entsprechen. Diese Eigenvektoren definieren die neuen Achsen des transformierten Datenraums. Hierfür wählen wir die besten n Komponenten in der Matrix $\boldsymbol{W}$.

```
# Sortiere die Eigenvektoren nach absteigenden Eigenwerten
idx = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:, idx]

# Wähle die n Hauptkomponenten
n_components = 2
eigenvalues_sub = eigenvalues[:n_components]
W = eigenvectors[:, :n_components]
```

Schritt 5 - Transformation der Daten: Die ursprünglichen Daten werden auf die neuen Hauptkomponenten projiziert, um die transformierten Daten zu erhalten.

$$\boldsymbol{X}_{\text{pca}} = \boldsymbol{X}_{\text{cent}} \cdot \boldsymbol{W}$$

```
# Transformiere die Daten
egywth_pca = np.dot(egywthCentered, W)
egywth_pca
```

```
array([[1764.35300097, -144.63239642],
       [ -70.67495979,  177.86671771],
       [1547.04272956,   99.44054371],
       ...,
       [ 920.66526878,  888.17908265],
       [ 914.15695   ,  895.99134945],
       [ 803.76433239, 1030.65117329]], shape=(916, 2))
```

PCA mit SciKit-Learn

Wichtig in der Praxis ist, dass PCA empfindlich gegenüber der Skalierung der Variablen ist. Variablen mit deutlich unterschiedlichen Skalen oder Größenordnungen können die ersten Hauptkomponenten sonst dominieren. Deshalb standardisiert man die Daten häufig vor einer PCA, wenn die Variablen unterschiedliche Einheiten oder Größenordnungen haben.

Für die Praxis gibt es dafür auch eine einfache Implementierung in `sklearn`, die wir wie gewohnt einsetzen. Wir können zum Training `fit` benutzen. Wollen wir allerdings gleich die Komponenten erhalten, bietet sich alternativ `fit_transform` an:

```
from sklearn.decomposition import PCA

pcam = PCA(n_components=2)
components = pcam.fit_transform(egywthNumber)
```

Wir können uns mit dem Attribut `explained_variance_` die Varianz der einzelnen Hauptkomponenten ausgeben lassen. Diese Werte entsprechen den Eigenwerten der Kovarianzmatrix. Das Attribut `singular_values_` enthält dagegen die Singulärwerte der zentrierten Datenmatrix aus der SVD, die `sklearn` intern zur Berechnung der PCA verwendet.

```
pcam.singular_values_
```

```
array([61197.4192215 , 11416.98508508])
```

Wir können uns auch mit `explained_variance_ratio_` die erklärte Varianz ausgeben lassen.

```
print(pcam.explained_variance_ratio_)
```

```
[0.96270602 0.0335066 ]
```

Hier sehen wir, dass die erste Komponente mit 96.7% den Großteil der Varianz erklärt, die zweite mit 3% deutlich weniger.

Als nächstes wollen wir die Komponenten visualisieren. Das ist ein weiterer wichtiger Anwendungsbereich von PCA. Da sich Datensätze mit sehr vielen Variablen schlecht visualisieren lassen, kann man sie mit PCA auf eine 2D- oder 3D-Darstellung reduzieren. Dies wird z.B. häufig beim Clustering verwendet, um höherdimensionale Cluster zu visualisieren.

```
import plotly.express as px
# Visualisierung der transformierten Daten
fig = px.scatter(components, x=0, y=1, width=500, height=500)
fig.update_layout( xaxis_title="Hauptkomponente 1",
yaxis_title="Hauptkomponente 2")
# Visualisierung der Eigenvektoren
```

```
loadings = pcam.components_.T * np.sqrt(pcam.explained_variance_)
for i, feature in enumerate(egywthCentered.columns):
    fig.add_annotation(ax=0, ay=0, axref="x", ayref="y", x=loadings[i, 0],
y=loadings[i, 1], showarrow=True, arrowsize=2, arrowhead=2, xanchor="right",
yanchor="top")
    fig.add_annotation( x=loadings[i, 0], y=loadings[i, 1], ax=0, ay=0,
xanchor="center", yanchor="bottom", text=feature, yshift=5 )
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

Welche Merkmale laden besonders stark auf die erste Hauptkomponente, und was sagt das über deren physikalische Bedeutung aus? Im Diagramm erkennen wir drei Gruppen um `"EV_NT_740"`, `"EV_HT_740"` und `"E_AV_Lab"`. Um letztere gruppieren sich auch die gesamten Wetterwerte. Das deutet darauf hin, dass diese deutlich stärker mit `"E_AV_Lab"` korreliert sind als mit den anderen Werten.

Nichtnegative Matrixfaktorisierung (NMF)

Methode

Matrixfaktorisierung wird genutzt, um die Anzahl der Beobachtungen (die Anzahl der Variablen bleibt gleich) zu reduzieren. Dabei zerteilt man einen Datensatz in zwei niedrigdimensionale Matrizen. Das ist sinnvoll, wenn man die Daten komprimieren oder latente Merkmale extrahieren möchte. Es wird z.B. bei der Bilderkennung benutzt, aber auch als Alternative zum Clustering, wenn keine klaren Cluster erkennbar sind.

Die *nichtnegative Matrixfaktorisierung* ist ein typischer Ansatz, bei dem alle Elemente der Matrizen nichtnegativ sind (sein müssen). Dies ist besonders nützlich für Anwendungen, bei denen die Daten nichtnegativ sind, wie z.B. Bilder, Texte und Empfehlungssysteme. Gegeben ist die nichtnegative Matrix $\boldsymbol{X}$ mit Dimension $m \times n$, die wir in die zwei Matrizen $\boldsymbol{W}$ und $\boldsymbol{H}$ mit den Dimensionen $m \times k$ und $k \times n$ zerlegen wollen, so dass

$$\boldsymbol{X} \approx \boldsymbol{W}\boldsymbol{H}.$$

wobei k die Anzahl der latenten Merkmale ist.

Wir wollen $\boldsymbol{W}$ und $\boldsymbol{H}$ finden, so dass die Differenz zwischen $\boldsymbol{X}$ und $\boldsymbol{W}\boldsymbol{H}$ minimiert wird. Die folgenden multiplikativen Aktualisierungsregeln minimieren den Rekonstruktionsfehler auf Basis der Frobenius-Norm [2]:

$$\min_{\boldsymbol{W},\boldsymbol{H}} \| \boldsymbol{X} - \boldsymbol{W}\boldsymbol{H} \|_F^2$$

Eine gebräuchliche Methode dafür ist der folgende iterative Algorithmus.

Schritt 1 - Initialisiere: Initialisiere $\boldsymbol{W}$ und $\boldsymbol{H}$ mit zufälligen nichtnegativen Werten.

Schritt 2 - Iterative Aktualisierung: Aktualisiere $\boldsymbol{W}$ und $\boldsymbol{H}$ iterativ unter Verwendung der multiplikativen Aktualisierungsregeln:

$$\boldsymbol{H} \leftarrow \boldsymbol{H} \odot \frac{\boldsymbol{W}^T \boldsymbol{X}}{\boldsymbol{W}^T \boldsymbol{W} \boldsymbol{H}}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} \odot \frac{\boldsymbol{X} \boldsymbol{H}^T}{\boldsymbol{W} \boldsymbol{H} \boldsymbol{H}^T}$$

wobei $\odot$ das elementweise Produkt und $\frac{A}{B}$ die elementweise Division bezeichnen. Neben dieser Variante auf Basis der Frobenius-Norm gibt es auch andere Zielfunktionen, z.B. Divergenzmaße für Zähldaten.

Schritt 3 - Konvergenzkriterium: Überprüfe die Konvergenz, z.B. ob die Änderung des Rekonstruktionsfehlers unter einen bestimmten Schwellenwert fällt oder eine maximale Anzahl von Iterationen erreicht ist.

Schritt 4 - Rekonstruktion: Die rekonstruierte Matrix $\hat{\boldsymbol{X}}$ ist das Produkt der Faktormatrizen $\boldsymbol{W}$ und $\boldsymbol{H}$:

$$\hat{\boldsymbol{X}} = \boldsymbol{W} \boldsymbol{H}.$$

NMF in SciKit-Learn

Auch für die nichtnegative Matrixfaktorisierung gibt es eine Implementierung in `sklearn`. Zu beachten ist, dass sie allerdings nur nichtnegative Werte verarbeiten kann. Deshalb müssen wir unsere Daten zuerst auf einen positiven Wertebereich *normalisieren*. Dafür können wir den `MinMaxScaler` verwenden, der unsere Daten auf den Wertebereich $0, \ldots, 1$ transformiert.

$$\boldsymbol{X}_{\text{mx}} = \frac{\boldsymbol{X} - \mathbf{min}_X}{\mathbf{max}_X - \mathbf{min}_X}.$$

mit dem Vektor der minimalen Werte $\mathbf{min}_X$ und maximalen Werte $\mathbf{max}_X$ aller Variablen in $\boldsymbol{X}$.

```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
egywthMinMax = mms.fit_transform(egywthNumber)
mms.min_, mms.scale_
```

```
(array([  0.        ,  -0.14705882,  -0.17349138,  -0.19689119,
          0.        ,  -0.14126394,  -0.08982036,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
         -0.13333333, -14.3641576 ,   0.1627907 ,  -0.55099483,
          0.09004739,   0.28315412,   0.415625  ]),
 array([0.00022624, 0.00022624, 0.00053879, 0.00259067, 0.00847458,
        0.03717472, 0.05988024, 0.02427184, 0.125     , 0.06198475,
```

```
        0.08333333, 0.125     , 0.05128205, 0.01470156, 0.02906977,
        0.01566661, 0.02369668, 0.03584229, 0.03125   ]))
```

Die Anwendung der NMF ist nun einfach.

```
from sklearn.decomposition import NMF

nmfm = NMF(n_components=2)
W = nmfm.fit_transform(egywthMinMax)
H = nmfm.components_
```

Hier gibt nun die Matrix W an, aus welchen Komponenten sich jede Beobachtung in unserem Datensatz zusammensetzt. Zum Beispiel sehen wir, dass sich die ersten zwei Zeilen nur aus der Komponente 1 zusammensetzen.

```
pd.DataFrame(W, columns=[f'Merkmal {i+1}' for i in range(n_components)])
```

	Merkmal 1	Merkmal 2
0	0.237477	0.000000
1	0.241596	0.000000
2	0.223237	0.016886
3	0.222422	0.028341
4	0.252235	0.000574
…	…	…
911	0.202307	0.057166
912	0.186173	0.083753
913	0.200195	0.037590
914	0.209245	0.025975
915	0.205023	0.028051

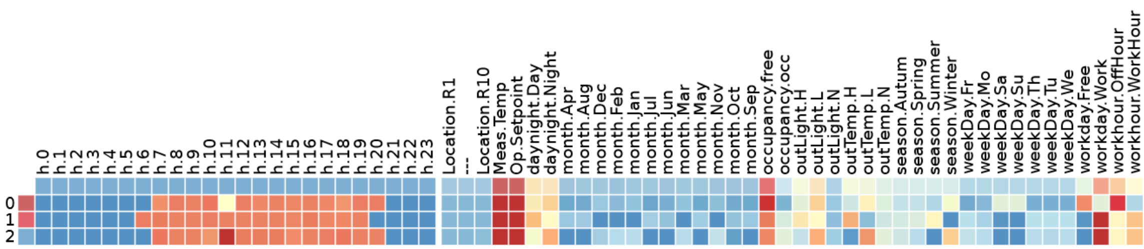
Interessanter ist die Matrix H, weil sie die extrahierten Komponenten darstellt. Hierfür transformieren wir sie zurück in den originalen Datenbereich, indem wir den MinMaxScaler mit `inverse_transform` anwenden.

```
H_rescaled=mms.inverse_transform(H)
pd.DataFrame(H_rescaled, columns=egywthNumber.columns)
```

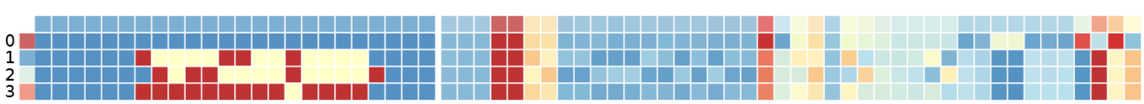
	EV_HT_7AE_LSV_Lab	EV_NE_7AE_LSV_Lab	FX	FM	RSK	RSKF	SHK_TAG	NM	VPM	PM	TMK	UPM	TXK	TNK	TGK
0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Diese zwei Komponenten sind nun zwei charakteristische Kombinationen, mit denen der Datensatz in dieser niedrigdimensionalen Darstellung approximiert wird. So lassen sich aus großen Datensätzen wichtige Muster ableiten. Man kann sie mit Vorsicht ähnlich wie typische Muster oder Prototypen auffassen.

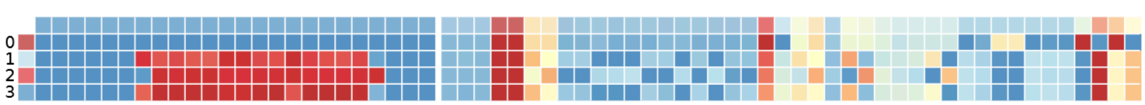
Die folgende Abbildung aus der Publikation „Understanding building operation from semantic context“ [3] zeigt das an einem Beispiel. In der Publikation werden die Steuerungsstrategien von Gebäuden aus Verbrauchsdaten abgeleitet. Die Abbildung vergleicht die Muster, die durch Clustering (k-Means), SiVM-Matrixfaktorisierung (vergleichbar mit NMF, die insbesondere charakteristische Muster identifiziert) und SiVM-Clustering (eine Kombination beider) identifiziert werden. Hierbei zeigt sich, dass die Muster, die durch SiVM-Matrixfaktorisierung identifiziert werden, deutlich ausgeprägter sind. Das liegt an der besonderen Methode *SiVM* zur Identifikation der Komponenten. Daraus wurden anschließend über *Assoziationsanalyse* Betriebsregeln abgeleitet, die Entscheidungsbäumen ähneln.



(b) Daily pattern extracted by k-means clustering



(c) Daily pattern extracted by SiVM matrix factorisation



(d) Daily pattern extracted by SiVM k-means clustering

Referenzen

Bibliographie

[1] K. Pearson, „On Lines and Planes of Closest Fit to Systems of Points in Space“, *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, Bd. 2, Nr. 11, S. 559–572, 1901, doi: 10.1080/14786440109462720.

[2] D. D. Lee und H. S. Seung, „Algorithms for Non-negative Matrix Factorization“, in *Advances in Neural Information Processing Systems 13*, 2001, S. 556–562.

[3] J. Ploennigs, B. Chen, und A. Schumann, „Understanding building operation from semantic context“, in *IECON - An. Conf. IEEE Ind. Elec. Soc.*, 2015.