KAPITEL 6

Erkunden

Man sagt, es sei leichter, einem Fachexperten die Wissenschaft von der Datenanalyse beizubringen als dem Datenanalysten das spezifische Fachwissen. Ich bin nicht sicher, ob ich da zu 100 % zustimme, aber es ist wahr, dass in Daten Feinheiten verborgen sind, die man mithilfe eines Fachexperten freilegen kann. Durch das gleichzeitige Verständnis von Fachgebiet und Datenanalyse können Fachexperten bessere Modelle erschaffen und damit ihr Fachgebiet voranbringen.

Bevor ich ein Modell erstelle, betreibe ich gewöhnlich ein wenig explorative Datenanalyse. Dadurch bekomme ich ein besseres Gespür für die Daten, aber es ist auch ein guter Anlass, sich mit den Fachleuten zusammenzusetzen, die die Kontrolle über die Daten haben, und mit ihnen Einzelheiten zu diskutieren.

Datenmenge

Wir verwenden hier ein weiteres Mal den Titanic-Datensatz. In pandas gibt die Eigenschaft `.shape` ein Tupel aus der Anzahl von Zeilen und Spalten zurück:

```
>>> X.shape
(1309, 13)
```

Wir sehen, dass dieser Datensatz 1.309 Zeilen und 13 Spalten umfasst.

Zusammenfassende Statistiken

Wir können mithilfe von pandas zusammenfassende Statistiken unserer Daten erheben. Die Methode `.describe` liefert uns darüber hinaus die Anzahl numerischer Werte (solche, die nicht NaN sind). Schauen wir uns die Ergebnisse für die erste und die letzte Spalte an:

```
>>> X.describe().iloc[:, [0, -1]]
            pclass   embarked_S
count  1309.000000  1309.000000
mean     -0.012831     0.698243
std       0.995822     0.459196
min      -1.551881     0.000000
```

```
25%      -0.363317      0.000000
50%       0.825248      1.000000
75%       0.825248      1.000000
max       0.825248      1.000000
```

Die Zeile »count« (Anzahl) sagt uns, dass beide Spalten ausgefüllt sind; es gibt keine fehlenden Werte. Wir sehen zudem Mittelwert, Standardabweichung, Minimum, Maximum und Quartile.

Ein DataFrame in pandas besitzt ein Attribut `iloc`, mit dessen Hilfe wir Indexoperationen ausführen können. Es erlaubt uns, anhand eines Index Zeilen und Spalten herauszugreifen. Wir geben die Positionen der Zeilen als Skalar, Liste oder Slice an und können dann, durch ein Komma getrennt, auch die Spalten per Skalar, Liste oder Slice auswählen.

In diesem Beispiel ziehen wir die zweite und die fünfte Zeile sowie die letzten drei Spalten heraus:

```
>>> X.iloc[[1, 4], -3:]
     sex_male  embarked_Q  embarked_S
677       1.0           0           1
864       0.0           0           1
```

Es gibt auch ein Attribut `.loc`, das Zeilen und Spalten über ihren Namen (statt der Position) herausgibt. Dies ist derselbe Ausschnitt aus dem DataFrame wie eben:

```
>>> X.loc[[677, 864], "sex_male":]
     sex_male  embarked_Q  embarked_S
677       1.0           0           1
864       0.0           0           1
```

Histogramm

Ein Histogramm eignet sich hervorragend, um numerische Daten zu veranschaulichen. Man sieht, wie viele Kategorien es gibt, und kann sich zugleich ihre Verteilung ansehen (siehe Abbildung 6-1). Die Bibliothek pandas stellt eine Methode `.plot` bereit, um Histogramme anzuzeigen:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> X.fare.plot(kind="hist", ax=ax)
>>> fig.savefig("images/mlpr_0601.png", dpi=300)
```

Mithilfe der Bibliothek seaborn können wir ein Histogramm stetiger Werte über der Zielgröße auftragen (siehe Abbildung 6-2):

```
fig, ax = plt.subplots(figsize=(12, 8))
mask = y_train == 1
ax = sns.distplot(X_train[mask].fare, label='survived')
ax = sns.distplot(X_train[~mask].fare, label='died')
ax.set_xlim(-1.5, 1.5)
ax.legend()
fig.savefig('images/mlpr_0602.png', dpi=300, bbox_inches='tight')
```

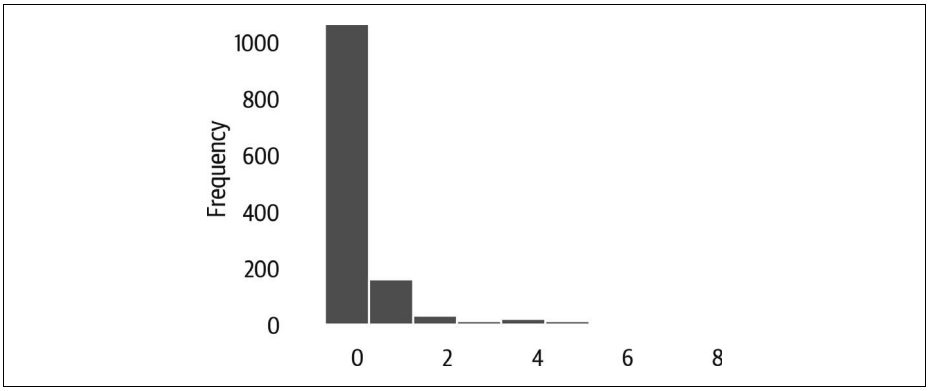

Abbildung 6-1: Histogramm mit pandas. Die vertikale Achse zeigt die Häufigkeit (Frequency) jedes Werts.

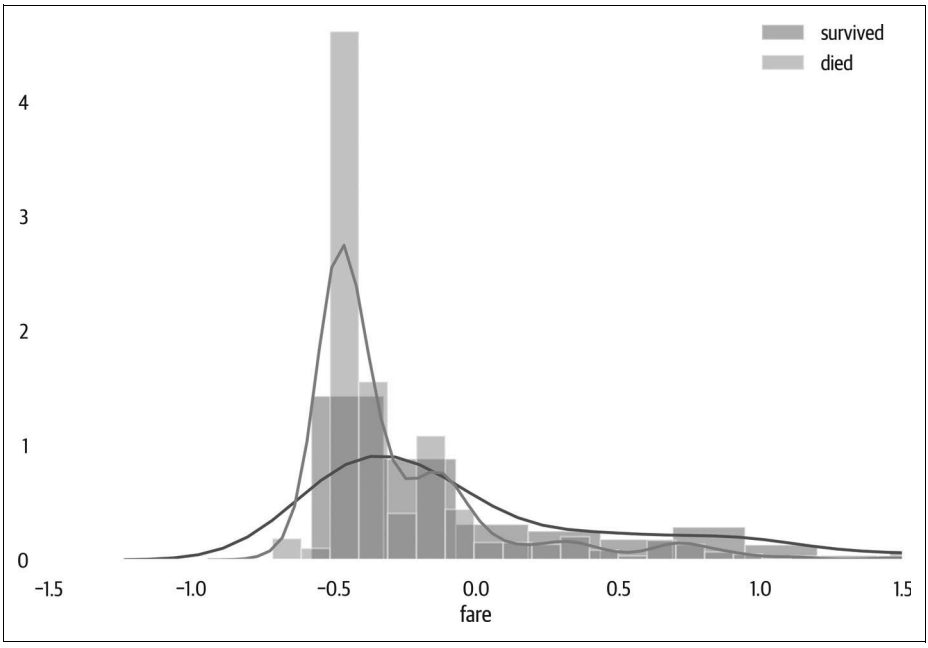

Abbildung 6-2: Histogramm mit seaborn

Streudiagramm

Ein Streudiagramm zeigt die Beziehung zwischen zwei numerischen Spalten (siehe Abbildung 6-3). Mit pandas geht das wieder sehr einfach. Wenn überlappende Daten vorkommen, passen Sie den Parameter `alpha` an:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> X.plot.scatter(
...     x="age", y="fare", ax=ax, alpha=0.3
... )
>>> fig.savefig("images/mlpr_0603.png", dpi=300)
```

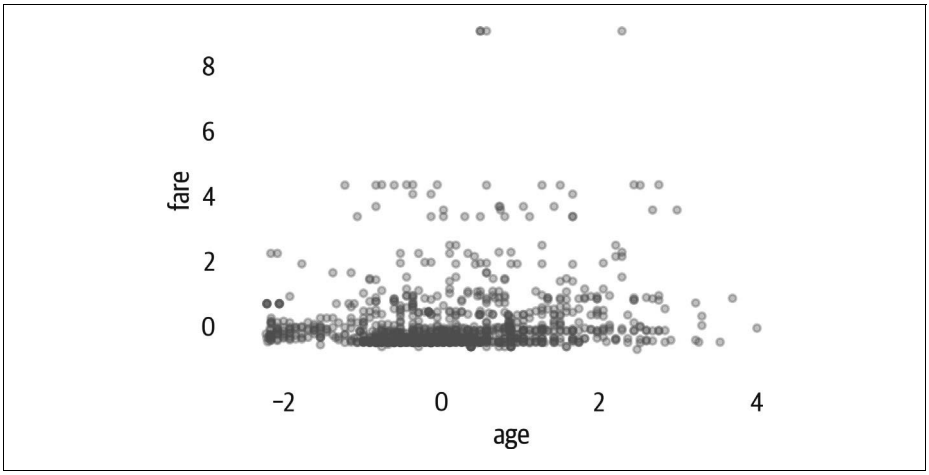

Abbildung 6-3: Streudiagramm mit pandas

Diese zwei Merkmale scheinen nicht besonders zu korrelieren. Wir können mittels der Methode `.corr` die Pearson-Korrelation zwischen zwei (pandas-)Spalten bestimmen, um die Korrelation zu beziffern:

```
>>> X.age.corr(X.fare)
0.17818151568062093
```

Kombidiagramm

Yellowbrick bietet ein aufwendigeres Streudiagramm an, das eine Regressionsgerade und an den Seiten zusätzlich Histogramme enthält und als *Kombidiagramm* (engl. Joint Plot) bezeichnet wird (siehe Abbildung 6-4):

```
>>> from yellowbrick.features import (
...     JointPlotVisualizer,
... )
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> jpv = JointPlotVisualizer(
...     feature="age", target="fare"
... )
>>> jpv.fit(X["age"], X["fare"])
>>> jpv.poof()
>>> fig.savefig("images/mlpr_0604.png", dpi=300)
```

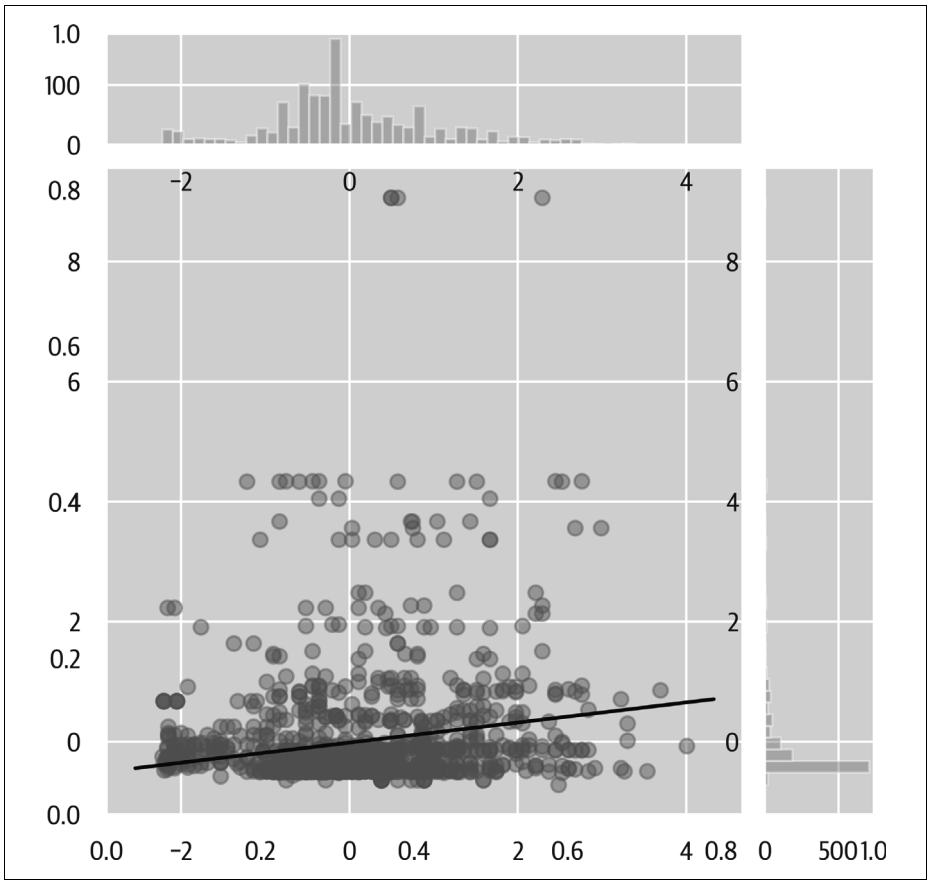

Abbildung 6-4: Kombidiagramm mit Yellowbrick

Je nachdem, mit welchen Argumenten der `JointPlotVisualizer` angelegt wurde, kann bei der Methode `.fit` sowohl der Parameter `X` als auch `y` eine Spalte bedeuten. Normalerweise ist `X` ein DataFrame und keine Folge von Werten.

Ein Kombidiagramm können Sie auch mit der Bibliothek seaborn (*https://seaborn.pydata.org*) erstellen (siehe Abbildung 6-5):

```
>>> from seaborn import jointplot
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> p = jointplot(
...     "age", "fare", data=new_df, kind="reg"
... )
>>> p.savefig("images/mlpr_0605.png", dpi=300)
```

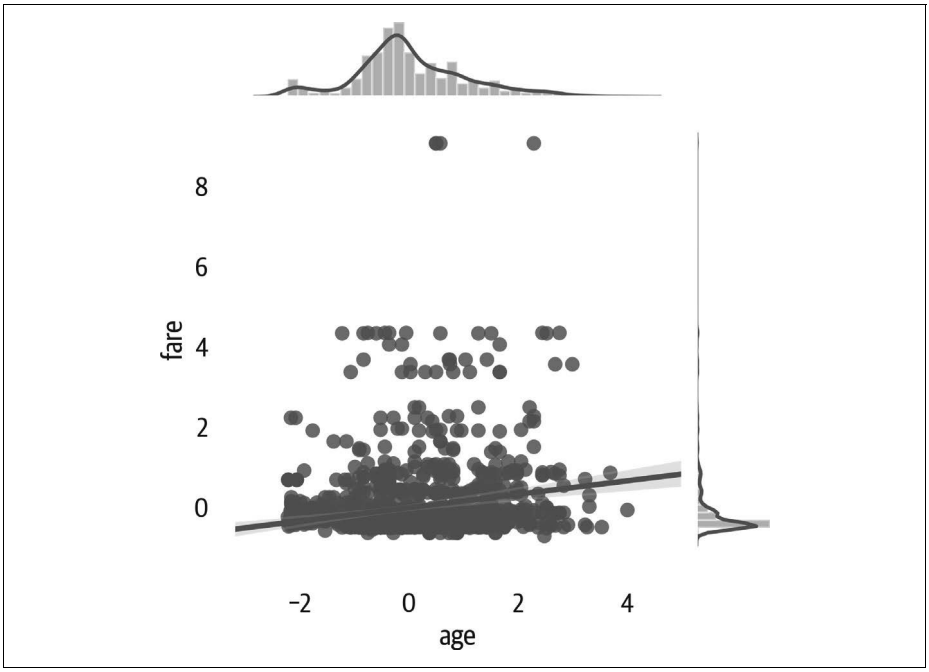

Abbildung 6-5: Kombidiagramm mit seaborn

Paarmatrix

Die Bibliothek seaborn kann eine Paarmatrix erzeugen (siehe Abbildung 6-6). Ein solches Diagramm ist eine Matrix von Spalten und Kerndichteschätzungen. Um es anhand einer Spalte aus einem DataFrame einzufärben, setzen Sie den Parameter `hue`. Färben wir anhand der Zielgröße ein, können wir sehen, ob sich bestimmte Merkmale unterschiedlich auf die Zielgröße auswirken:

```
>>> from seaborn import pairplot
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> vars = ["pclass", "age", "fare"]
>>> p = pairplot(
...     new_df, vars=vars, hue="target", kind="reg"
... )
>>> p.savefig("images/mlpr_0606.png", dpi=300)
```

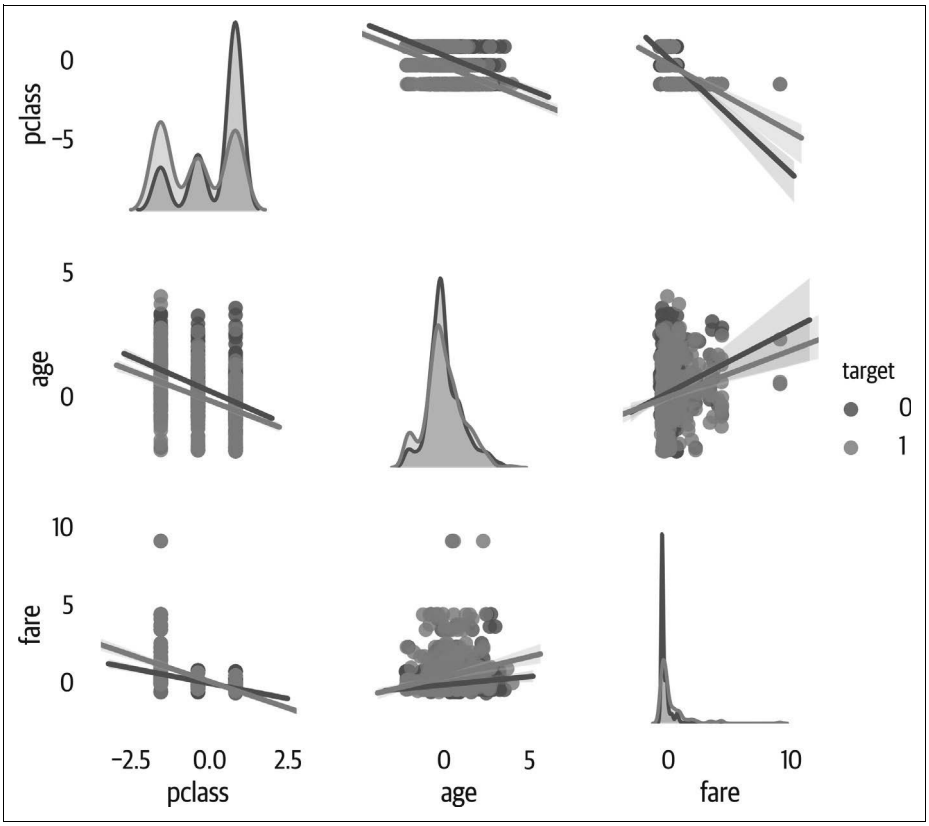

Abbildung 6-6: Paarmatrix mit seaborn für eine zweiwertige Zielgröße (Target)

Kasten- und Violinendiagramme

Seaborn kennt unterschiedliche Arten von Diagrammen, um Verteilungen zu veranschaulichen. Wir zeigen je ein Beispiel eines Kasten- und eines Violinendiagramms (siehe Abbildung 6-7 und Abbildung 6-8). Diese Diagramme können ein Merkmal in Abhängigkeit von einer Zielgröße darstellen:

```
>>> from seaborn import box plot
>>> fig, ax = plt.subplots(figsize=(8, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> boxplot(x="target", y="age", data=new_df)
>>> fig.savefig("images/mlpr_0607.png", dpi=300)
```

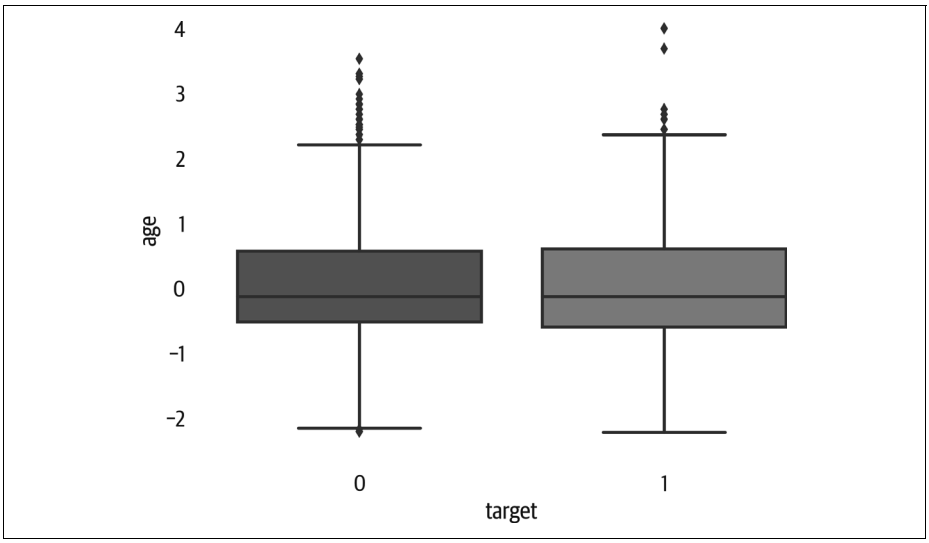

Abbildung 6-7: Kastendiagramm mit seaborn für eine zweiwertige Zielgröße (Target)

Violinendiagramme sind hilfreich, um Verteilungen zu veranschaulichen:

```
>>> from seaborn import violinplot
>>> fig, ax = plt.subplots(figsize=(8, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> violinplot(
...     x="target", y="sex_male", data=new_df
... )
>>> fig.savefig("images/mlpr_0608.png", dpi=300)
```

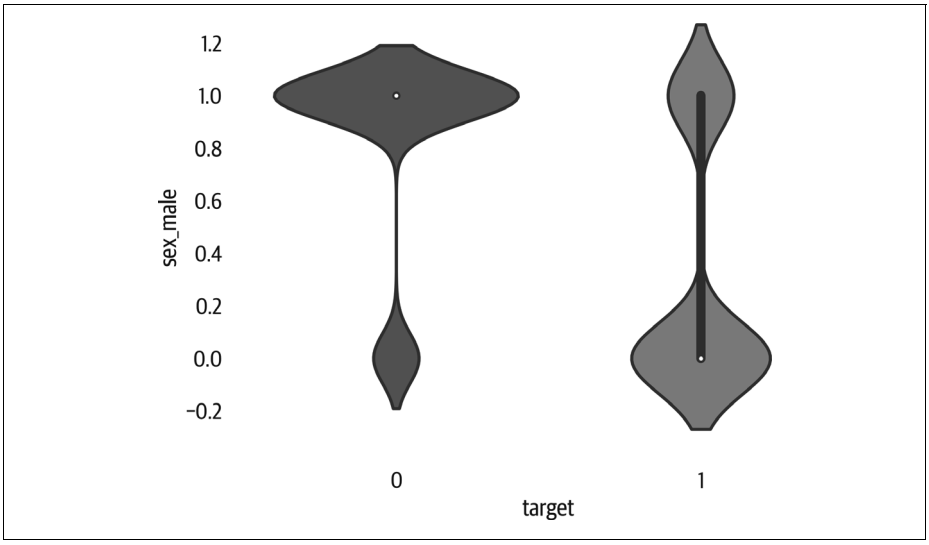

Abbildung 6-8: Violinendiagramm mit seaborn für eine zweiwertige Zielgröße (Target)

Vergleich zweier Ordinalwerte

Der folgende pandas-Code vergleicht zwei Ordinalkategorien. Um so etwas zu simulieren, fasse ich Altersangaben zu zehn Quantilen und Passagierklassen in drei Klassen zusammen. Das Diagramm ist normalisiert, sodass es die gesamte Höhe ausfüllt. Auf diese Weise sieht man leicht, dass sich die meisten Fahrkarten im 40-%-Quantil für die dritte Klasse befanden (siehe Abbildung 6-9):

```
>>> fig, ax = plt.subplots(figsize=(8, 6))
>>> (
...     X.assign(
...         age_bin=pd.qcut(
...             X.age, q=10, labels=False
...         ),
...         class_bin=pd.cut(
...             X.pclass, bins=3, labels=False
...         ),
...     )
...     .groupby(["age_bin", "class_bin"])
...     .size()
...     .unstack()
...     .pipe(lambda df: df.div(df.sum(1), axis=0))
...     .plot.bar(
...         stacked=True,
...         width=1,
...         ax=ax,
...         cmap="viridis",
...     )
...     .legend(bbox_to_anchor=(1, 1))
... )
>>> fig.savefig(
...     "image/mlpr_0609.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

Die Zeilen

```
.groupby(["age_bin", "class_bin"])
.size()
.unstack()
```

können durch Folgendes ersetzt werden:

```
.pipe(lambda df: pd.crosstab(
      df.age_bin, df.class_bin)
)
```

In pandas gibt es oft mehr als einen Weg zum Ziel, und es stehen einige Hilfsfunktionen zur Verfügung, die andere Funktionen zusammenfügen, beispielsweise `pd.crosstab`.

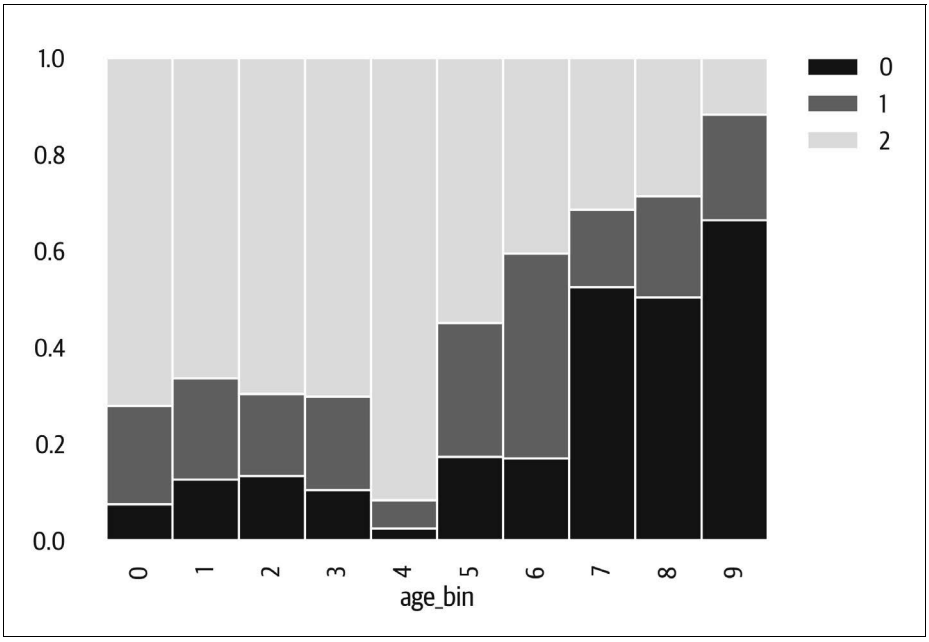

Abbildung 6-9: Vergleich von Ordinalwerten

Korrelation

Yellowbrick kann Merkmale paarweise vergleichen (siehe Abbildung 6-10). Das folgende Diagramm zeigt eine Pearson-Korrelation (der Parameter `algorithm` kann auch die Werte `'spearman'` oder `'covariance'` annehmen):

```
>>> from yellowbrick.features import Rank2D
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> pcv = Rank2D(
...     features=X.columns, algorithm="pearson"
... )
>>> pcv.fit(X, y)
>>> pcv.transform(X)
>>> pcv.poof()
>>> fig.savefig(
...     "images/mlpr_0610.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

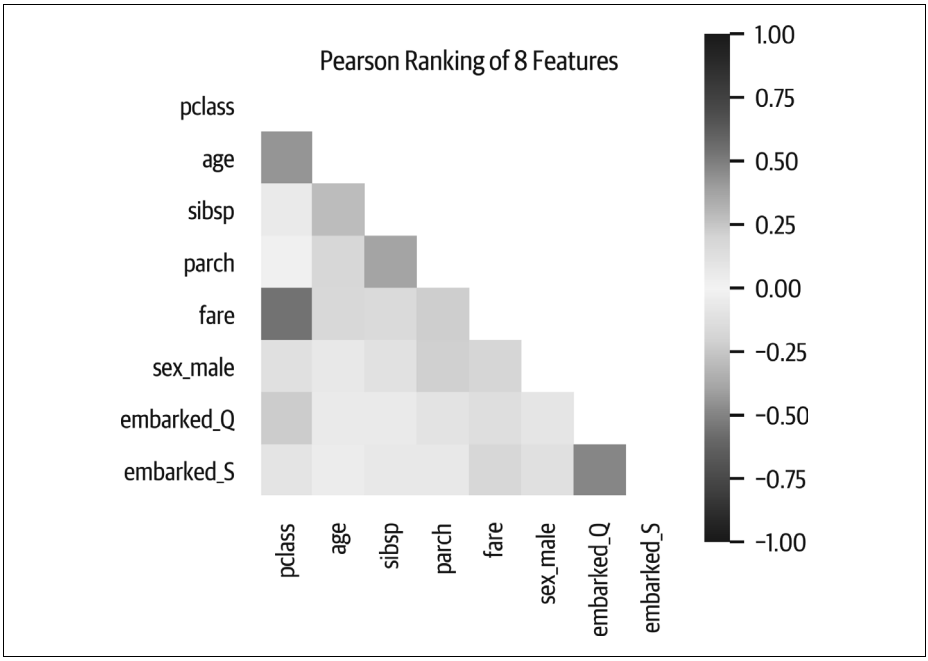

Abbildung 6-10: Pearson-Korrelation mit Yellowbrick für acht Merkmale (Features)

Ein ähnliches Diagramm, eine Heatmap, wird von der Bibliothek seaborn angeboten (siehe Abbildung 6-11). Als Daten müssen wir einen DataFrame mit der Korrelation übergeben. Damit der Farbumfang auch dann von –1 bis 1 reicht, wenn die Daten in der Matrix nicht diesen Wertebereich haben, fügen wir die Parameter `vmin` und `vmax` hinzu:

```
>>> from seaborn import heatmap
>>> fig, ax = plt.subplots(figsize=(8, 8))
>>> ax = heatmap(
...     X.corr(),
...     fmt=".2f",
...     annot=True,
...     ax=ax,
...     cmap="RdBu_r",
...     vmin=-1,
...     vmax=1,
... )
>>> fig.savefig(
...     "images/mlpr_0611.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

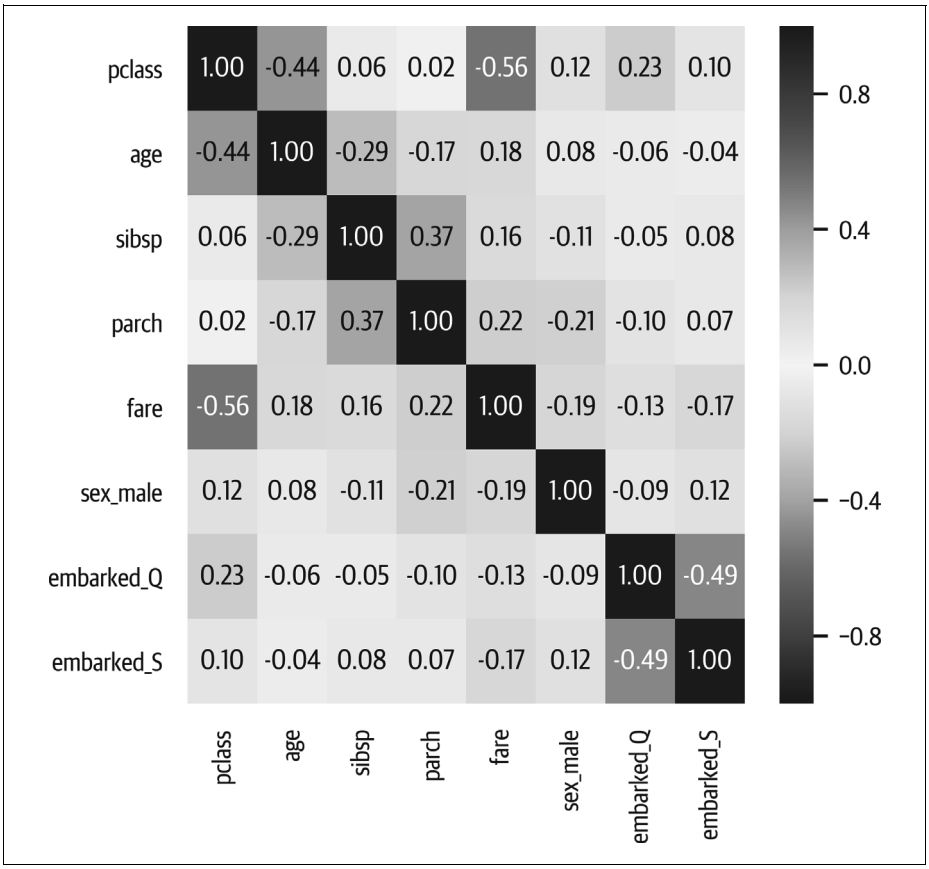

Abbildung 6-11: Heatmap mit seaborn

Die Bibliothek pandas kann die Korrelation zwischen Spalten in einem DataFrame bestimmen. Wir zeigen hier nur die ersten zwei Spalten des Ergebnisses. Von Haus aus wird der Algorithmus `'pearson'` benutzt, aber Sie können den Parameter `method` auch auf `'kendall'`, `'spearman'` oder ein eigenes Callable setzen, das für zwei Spalten eine Fließkommazahl zurückgibt:

```
>>> X.corr().iloc[:, :2]
              pclass       age
pclass      1.000000 -0.440769
age        -0.440769  1.000000
sibsp       0.060832 -0.292051
parch       0.018322 -0.174992
fare       -0.558831  0.177205
sex_male    0.124617  0.077636
embarked_Q  0.230491 -0.061146
embarked_S  0.096335 -0.041315
```

Stark korrelierte Spalten tragen keinen Mehrwert bei und können die Gewichtung von Merkmalen und die Interpretation der Regressionskoeffizienten verfälschen. Der nachstehende Code findet korrelierte Spalten. In unseren Daten gibt es allerdings keine stark korrelierten Spalten (schließlich hatten wir ja die Spalte »sex_male« entfernt).

Hätten wir jedoch korrelierte Spalten, könnten wir entweder die Spalten aus level_0 oder die aus level_1 aus den Merkmalsdaten entfernen:

```
>>> def correlated_columns(df, threshold=0.95):
...     return (
...         df.corr()
...         .pipe(
...             lambda df1: pd.DataFrame(
...                 np.tril(df1, k=-1),
...                 columns=df.columns,
...                 index=df.columns,
...             )
...         )
...         .stack()
...         .rename("pearson")
...         .pipe(
...             lambda s: s[
...                 s.abs() > threshold
...             ].reset_index()
...         )
...         .query("level_0 not in level_1")
...     )

>>> correlated_columns(X)
Empty DataFrame
Columns: [level_0, level_1, pearson]
Index: []
```

Dem Datensatz mit mehr Spalten können wir ansehen, dass viele davon korrelieren:

```
>>> c_df = correlated_columns(agg_df)
>>> c_df.style.format({"pearson": "{:.2f}"})
       level_0     level_1   pearson
3   pclass_mean      pclass  1.00
4   pclass_mean  pclass_min  1.00
5   pclass_mean  pclass_max  1.00
6    sibsp_mean   sibsp_max  0.97
7    parch_mean   parch_min  0.95
8    parch_mean   parch_max  0.96
9     fare_mean        fare  0.95
10    fare_mean    fare_max  0.98
12    body_mean    body_min  1.00
13    body_mean    body_max  1.00
14     sex_male  sex_female -1.00
15   embarked_S  embarked_C -0.95
```

RadViz

Ein RadViz-Diagramm ordnet jeden Datenpunkt in einem Kreis an, auf dessen Rand sich die Merkmale befinden (siehe Abbildung 6-12). Die Werte sind normalisiert, und man kann sich vorstellen, dass jeder Merkmalspunkt Datenpunkte entsprechend dem jeweiligen Wert wie an einer Spiralfeder zu sich zieht.

Hierbei handelt es sich um eine mögliche Technik, die Unterscheidbarkeit zwischen den Zielgrößen anschaulich darzustellen.

Mit Yellowbrick sieht das folgendermaßen aus:

```
>>> from yellowbrick.features import RadViz
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> rv = RadViz(
...     classes=["died", "survived"],
...     features=X.columns,
... )
>>> rv.fit(X, y)
>>> _ = rv.transform(X)
>>> rv.poof()
>>> fig.savefig("images/mlpr_0612.png", dpi=300)
```

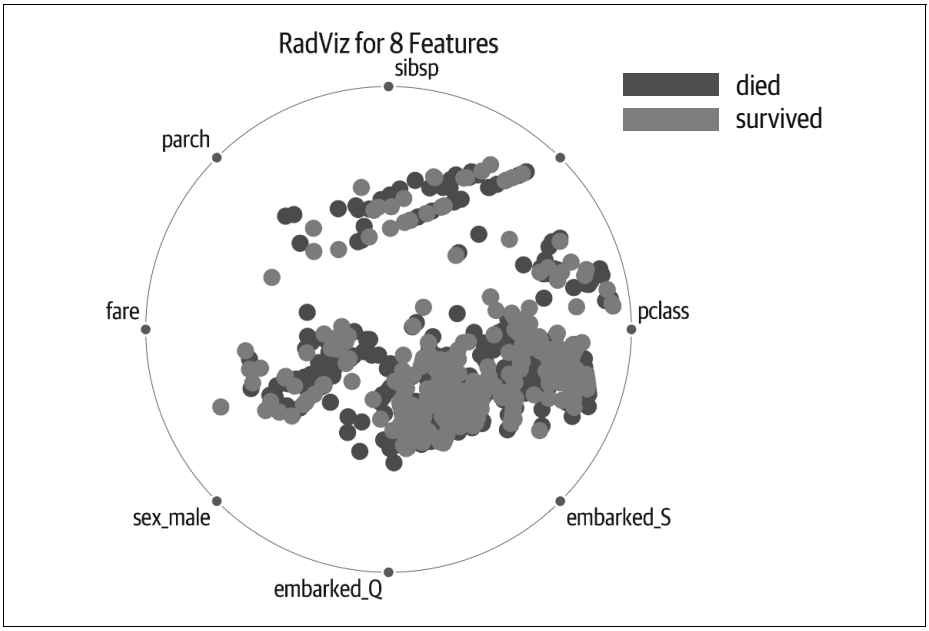

Abbildung 6-12: RadViz-Diagramm mit Yellowbrick für acht Merkmale (Features)

Die Bibliothek pandas kann ebenfalls RadViz-Diagramme zeichnen (siehe Abbildung 6-13):

```
>>> from pandas.plotting import radviz
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> radviz(
...     new_df, "target", ax=ax, colormap="PiYG"
... )
>>> fig.savefig("images/mlpr_0613.png", dpi=300)
```

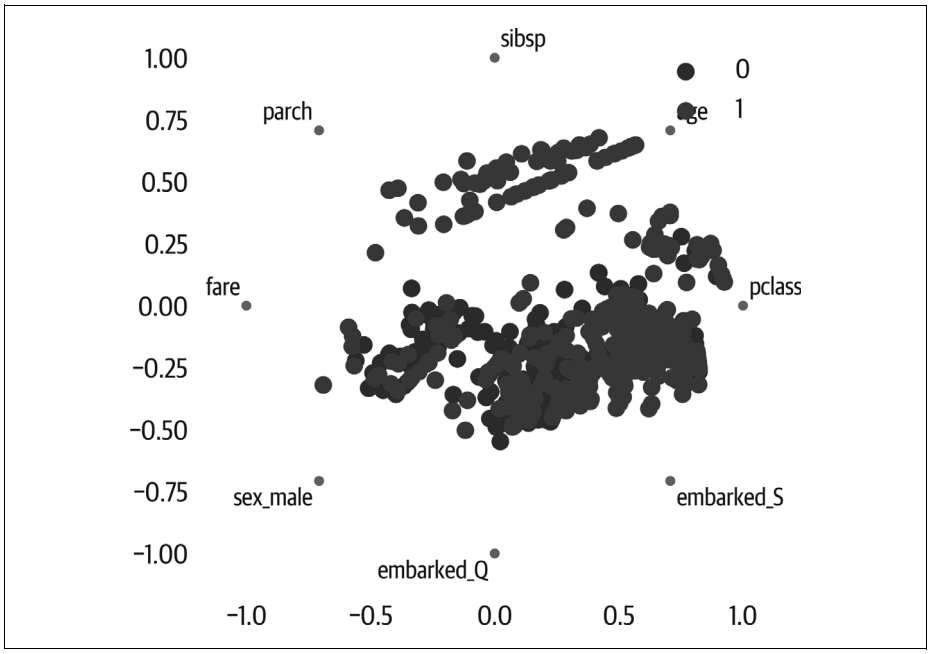

Abbildung 6-13: RadViz-Diagramm mit pandas

Parallele Koordinaten

Bei multivariaten Daten lassen sich Cluster mithilfe paralleler Koordinaten sichtbar machen (siehe Abbildung 6-14 und Abbildung 6-15).

Es folgt wieder eine Version mit Yellowbrick:

```
>>> from yellowbrick.features import (
...     ParallelCoordinates,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pc = ParallelCoordinates(
...     classes=["died", "survived"],
...     features=X.columns,
... )
```

```
>>> pc.fit(X, y)
>>> pc.transform(X)
>>> ax.set_xticklabels(
...     ax.get_xticklabels(), rotation=45
... )
>>> pc.poof()
>>> fig.savefig("images/mlpr_0614.png", dpi=300)
```

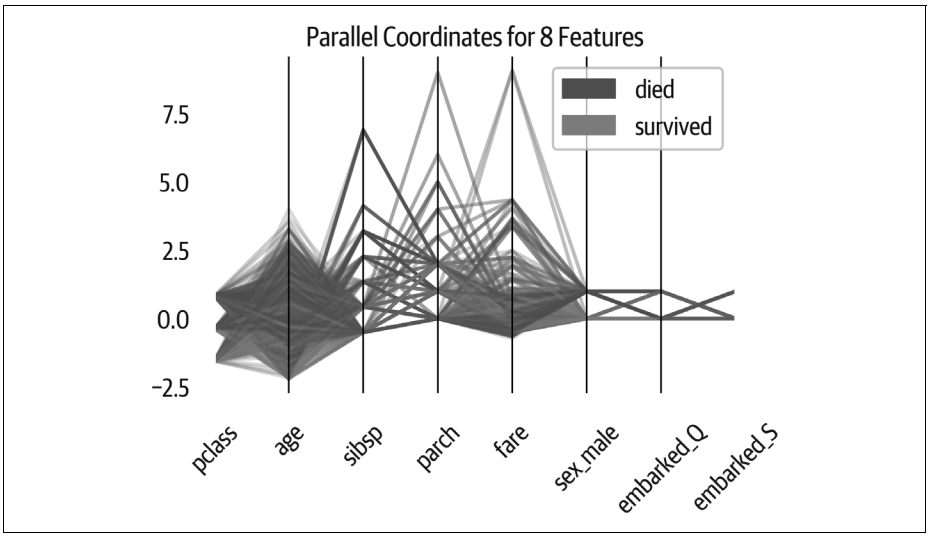

Abbildung 6-14: Parallele Koordinaten mit Yellowbrick für acht Merkmale (Features)

Hier noch eine pandas-Version:

```
>>> from pandas.plotting import (
...     parallel_coordinates,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> parallel_coordinates(
...     new_df,
...     "target",
...     ax=ax,
...     colormap="viridis",
...     alpha=0.5,
... )
>>> ax.set_xticklabels(
...     ax.get_xticklabels(), rotation=45
... )
>>> fig.savefig(
...     "images/mlpr_0615.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

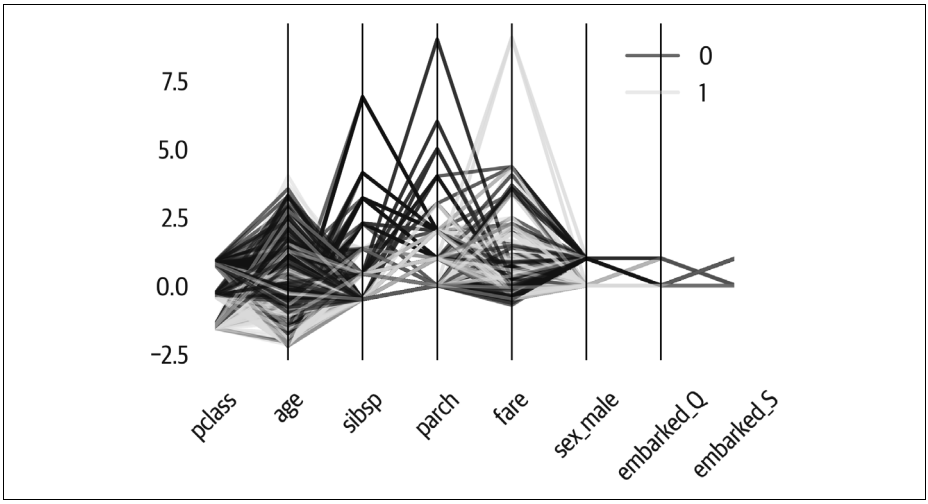

Abbildung 6-15: Parallele Koordinaten mit pandas